嵌入式Linux
驱动程序设计
从入门到精通

冯国进　编著

附光盘

- 各章驱动范例代码、内核代码
- 本书所有芯片资料、标准文档
- 驱动开发交叉编译环境

清华大学出版社

VISIT…

LANZAROTE Caliente.com

嵌入式 Linux 驱动程序设计
从入门到精通

冯国进　编著

清华大学出版社
北　京

内 容 简 介

本书基于 Linux 2.6 内核讲述了 Linux 嵌入式驱动程序开发的知识，全书内容涵盖了 Linux 2.6 下的三类驱动设备，包括 Linux 下字符设备、块设备、网络设备的开发技术。具体内容包括 Linux 驱动开发入门基础知识，Linux 操作系统下驱动开发核心技术，并对 ARM 系统的各类接口的原理、驱动开发与应用层开发进行逐一分析，其中包括 GPIO、CAN、I^2C、LCD、USB、触摸屏、网络、块设备、红外、SD 卡等接口。

本书主要面向嵌入式 Linux 系统的内核、驱动和应用程序的开发人员以及 ARM 嵌入式系统的接口设计人员，可以作为各类嵌入式系统培训机构和高校操作系统课程的实验教材和辅导书籍。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989 13501256678 13801310933

图书在版编目（CIP）数据

嵌入式 Linux 驱动程序设计从入门到精通 / 冯国进编著. —北京：清华大学出版社，2008.3
ISBN 978-7-302-16942-0

Ⅰ. 嵌… Ⅱ. 冯… Ⅲ. Linux 操作系统–程序设计 Ⅳ. TP316.89

中国版本图书馆 CIP 数据核字（2008）第 012807 号

责任编辑： 夏兆彦
责任校对： 张 剑
责任印制：
出版发行： 清华大学出版社　　**地　址：** 北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：** 100084
c-service@tup.tsinghua.edu.cn
社 总 机： 010-62770175　　**邮购热线：** 010-62786544
投稿咨询： 010-62772015　　**客户服务：** 010-62776969
印 刷 者：
装 订 者：
经　销： 全国新华书店
开　本： 203×260　**印张：** 20.5　　**字数：** 531 千字
版　次： 2008 年 3 月第 1 版　　**印次：** 2008 年 3 月第 1 次印刷
印　数： 1~ 000
定　价： 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。
联系电话：010-62770177 转 3103　产品编号：027613-01

FOREWORD

序言

随着现代智能设备的不断升级换代，基于 ARM 等嵌入式系统等的 32 位机智能系统在现代生活中的地位也越来越重要。我也经常上书店，期望寻觅一些比较贴近某一实际系统开发，更有参考价值与使用意义的书籍，可是发现除了部分针对 ARM9 系统开发板的简单介绍外，少有比较系统全面地介绍驱动开发的书籍，这一现实情况给科研工作者的实际工作带来不少困难。我周围就有不少科研人员从事该项技术研究，由于不太了解更深层次的技巧与方法，有时遇到问题不得不设法与芯片供应商的技术人员进行沟通，但问题往往还是不能得到及时圆满的解决。

冯国进和我提到正在编写一本嵌入式 linux 驱动开发方面的书，同时也向我介绍了此书的主要内容和组织结构，并嘱我为此书作序，我欣然同意。纵观本书，其内容涵盖了 linux2.6 下的三类驱动设备，包括 linux 下字符设备、块设备、网络设备的开发技术。本书全面地分析了嵌入式 linux 下驱动开发的核心技术，并深入探讨了 ARM 嵌入式系统各类接口的原理、驱动开发与应用层开发技术，其内容不禁深深吸引了我，相信该书将会对从事嵌入式系统研究的科研人员有极大的帮助。

冯国进在攻读硕士学位期间就对软件开发、图像处理算法和硬件设计显示出了浓厚的兴趣和独有的天赋，同时在科研学术上又非常执着与严谨，在此期间他所做的工作在我们学校科研工作中发挥了重要作用，至今我仍常向我现在的学生提起他，并希望他们能从他身上学到点什么。冯国进硕士毕业后一直致力于嵌入式系统的开发研究，应该说本书是其多年研究工作的经验总结和结晶，各个功能模块基本都是经他亲自调试验证过了的。

学习、研究过程的一个重要环节是总结，本书就是一个嵌入式系统开发研究人员的总结。我也希望更多的科研人员在学习此书、从事该类技术研究的过程中能多一些像这样的总结。

借此机会，向出版此书的清华大学出版社表示感谢，同时，也向承担此书编辑、整理、审订等浩繁工作的所有人员表示敬意。

顾国华

南京理工大学电子工程与光电技术学院　教授

2007-11-3

FOREWORD

前言

我学编程是从 VB 开始的，看的第一本编程书有《红楼梦》那么厚，现在想来仍然可怕。从那时开始我与软件开发便结下了不解之缘。后来我逐渐转向 VC++编程。三年前有幸接触到 Linux 操作系统开发，才真正领略到软件艺术的迷人风采，并真正喜欢上软件开发工作。兴趣是最好的老师。在工作中，我常常需要自己开发 Linux 下的驱动程序，但是国内市场上关于驱动开发的书籍寥若晨星，网络上的资料又不全面，不成系统。于是我萌生了写一本驱动开发方面的书籍的念头，希望能够稍微揭开一点点 Linux 驱动开发的神秘面纱，一方面是让更多的人少走弯路，为中国的软件业做一点微不足道的贡献；更重要的是抛砖引玉，希望以后有更多的同类书籍问世。

内核版本：

为什么选择 Linux，开源无疑是最大的诱惑。很多程序员喜欢 Linux，因为 Linux 让他们掌握了自己的命运。Linux 最初是在网络应用方面比较占优势，后来逐渐在嵌入式系统中占有一席之地，成为能与 Wince、Vxworks 抗衡的高端嵌入式操作系统之一，受众多大公司的欢迎，具有广阔的发展潜力。本书基于 Linux 2.6 内核。相对于 Linux 2.4 内核，Linux 2.6 内核做了相当大地改进，修改了驱动模型，增加了对更多设备的支持。

硬件平台：

随着人们生活水平的提高，人们对智能产品的需求越来越高。很多产品用 8 位机已经很难满足要求，我们已经迎来了 32 位应用的时代。目前在 32 位机市场上，基于 ARM 的嵌入式市场目前的出货量每年超过 15 亿，ARM 系统在业界拥有最多的第三方工具和解决方案供应商。ARM 公司自 1990 年正式成立以来，在 32 位 RISC（Reduced Instruction Set Computer）CPU 开发领域不断取得突破，其结构已经从 V3 发展到 V6。目前非常流行的 ARM 芯核有 ARM7TDMI，ARM720T，ARM9TDMI，ARM920T，ARM940T，ARM926EJ-S，ARM1020E 和 XScale 等。本书基于三星公司的 ARM920T 处理器 S3C2410X。

组织结构：

本书涵盖了 Linux 2.6 下的三类驱动设备，包括 Linux 下字符设备、块设备、网络设备的开发技术。第 1 章为 Linux 驱动开发入门基础知识。第 2 章讲述 Linux 操作系统下驱动开发核心技术。第 3～11 章对 ARM 系统的各类接口的原理、驱动开发与应用层开发进行逐一分析，其中包括 GPIO、CAN、I^2C、LCD、USB、触摸屏、网络、块设备、红外、SD 卡等接口。

读者对象：

本书是一本专门介绍嵌入式 Linux 驱动开发的书籍。本书主要面向嵌入式 Linux 系统的内核与驱动和应用程序的开发人员以及 ARM 嵌入式系统的接口设计人员，也可以作为各类嵌入式系统培训机构的培训实验教材和高校操作系统课程的辅导书籍。

光盘内容:

本书所附的光盘中包含本书各章代码、一些芯片资料和驱动开发文档。所有代码均在 YL2410 开发板上实验通过。

特别致谢：

特别感谢深圳优龙科技提供的 YL2410 开发板，他们的专业精神为嵌入式开发人员奉献了许多优秀的开发板。由于 Linux 下的驱动开发相当复杂，加之本人水平有限，错误在所难免，敬请各位读者谅解并指正，以便再版时修正。

冯国进
2007 年 10 月

CONTENTS

目录

第 1 章　Linux 驱动程序基础 …… 1
1.1　驱动程序的概念 …… 1
1.2　Linux 驱动程序模型 …… 1
1.3　最基本的调试手段 …… 5
1.4　导出符号的方法 …… 5
1.5　动态加载驱动程序 …… 6
1.6　在内核中加入新驱动 …… 6
1.7　应用程序操作接口 …… 7
1.8　第一个驱动 …… 10

第 2 章　Linux 驱动开发核心技术 …… 17
2.1　同步机制 …… 17
2.1.1　自旋锁 …… 17
2.1.2　信号量 …… 18
2.1.3　原子操作 …… 18
2.1.4　读写锁（rwlock） …… 19
2.1.5　seqlock 机制 …… 21
2.1.6　RCU …… 22
2.2　完成事件 …… 24
2.3　阻塞与非阻塞 …… 25
2.4　时间 …… 27
2.4.1　Linux 下延迟 …… 27
2.4.2　内核定时器 …… 27
2.5　内存分配与映射 …… 28
2.5.1　内存分配与释放 …… 28
2.5.2　用户态和内核态内存交互 …… 29
2.5.3　内存池 …… 30
2.5.4　物理地址到虚拟地址的映射 …… 31
2.5.5　内核空间到用户空间的映射 …… 31
2.6　中断处理 …… 32
2.6.1　硬件中断 …… 32
2.6.2　软中断机制 …… 35
2.7　/proc 系统 …… 36
2.8　工作队列 …… 38
2.9　异步 I/O …… 39
2.10　DMA …… 42

2.11 platform 概念 43
2.12 简单驱动例程 45
2.12.1 信号量同步 45
2.12.2 阻塞式读写 46
2.12.3 定时器 48
2.12.4 内存映射 49
2.12.5 /proc 访问 53
2.12.6 工作队列 55

第 3 章 GPIO 驱动 57
3.1 ARM 体系结构概述 57
3.1.1 RISC 结构 57
3.1.2 处理器模式 58
3.1.3 寄存器组织 58
3.1.4 异常处理 60
3.2 S3C2410X 处理器 61
3.3 S3C2410X I/O 端口 63
3.4 最简单的设备驱动——LED 灯驱动 64
3.5 S3C2410X GPIO 键盘驱动 66

第 4 章 串行总线驱动 73
4.1 串行总线综述 73
4.1.1 I^2C 总线 73
4.1.2 SMBus 总线 75
4.1.3 SPI 总线 76
4.1.4 CAN 总线 76
4.2 CAN 接口芯片 MCP2510 79
4.2.1 数据发送 79
4.2.2 数据接收 81
4.2.3 中断 83
4.2.4 波特率设置 84
4.2.5 工作模式 85
4.3 MCP2510 驱动开发 86
4.4 Linux 的 I^2C 驱动架构 96
4.5 Linux I^2C 驱动开发 100

第 5 章 USB 驱动程序 102
5.1 USB 总线 102
5.1.1 USB 总线概述 102
5.1.2 USB 系统组成 102
5.1.3 USB 传输模式 104
5.1.4 主机规范 105
5.1.5 USB 设备描述符 105
5.1.6 HID 类规范 110
5.2 Linux 下的 USB 驱动框架 111
5.3 USB 请求块 urb 114
5.4 USB 骨架程序 118
5.5 USB 文件系统 126
5.6 USB 摄像头驱动 127
5.6.1 USB 摄像头原理 127
5.6.2 Video4Linux 规范 128
5.6.3 OV511 驱动分析与编译 132
5.6.4 spca5xx 编译与使用 139
5.7 USB Gadget 140
5.7.1 USB 设备控制器驱动 142
5.7.2 Gadget 驱动 146

第 6 章 Linux Framebuffer 驱动 150
6.1 LCD 原理 150
6.2 Linux 下 LCD 驱动架构 151
6.3 S3C2410X LCD 控制器 157
6.4 S3C2410X LCD 驱动开发 163
6.5 基于 Framebuffer 的界面系统开发 168

第 7 章 输入子系统驱动 174
7.1 Linux 输入设备驱动 174
7.2 键盘输入设备驱动 179
7.3 在 MiniGUI 中加入键盘驱动 184
7.4 LED 输入设备驱动 188
7.5 USB 鼠标输入设备驱动 190

第 8 章 触摸屏驱动 196
8.1 触摸屏原理 196
8.2 S3C2410X 触摸屏控制器 197
8.3 S3C2410X 触摸屏驱动设计 200
8.4 校准原理及编程思路 204
8.4.1 线性校准原理 205
8.4.2 三点校准原理 205

8.5 利用 tslib 库校准 207
8.6 在 MiniGUI 中加入触摸屏驱动 211

第 9 章 块设备驱动 213
9.1 Linux 块设备驱动 213
9.2 简单块设备驱动 217
9.3 Linux 文件系统 221
9.4 MTD 驱动分析 223
9.5 cramfs 文件系统 224
9.6 NAND 和 NOR Flash 225
9.7 在系统中添加 JFFS2 分区 226

第 10 章 SD 卡驱动 229
10.1 SD 卡概述 229
10.2 SD 卡的通信 231
10.3 SD 卡寄存器 233
10.4 Linux 对 SD 卡的支持 235
10.4.1 重要数据结构 236
10.4.2 MMC/SD 卡块设备驱动 238
10.4.3 SD 卡扫描 243
10.5 如何开发一个 SD 驱动 244

第 11 章 网络设备驱动 249
11.1 网络驱动基础 249
11.2 sk_buff 253
11.3 Linux 网络设备驱动架构 256
11.4 一个虚拟网络设备驱动 259
11.5 DM9000 网卡芯片 262
11.6 DM9000 网卡驱动程序分析 265

第 12 章 红外设备驱动 276
12.1 红外通信协议规范 276
12.2 S3C2410X 红外接口 277
12.3 S3C2410X 红外设备驱动 279
12.4 Linux 对红外网络通信的支持 282
12.5 红外 SOCKET 通信 285

第 13 章 音频设备驱动 291
13.1 Linux 音频体系 291
13.2 UDA1341TS 音频原理 292
13.3 S3C2410X 的音频接口 294
13.4 UDA1341TS 驱动开发 302
13.5 音频应用层编程 308
13.5.1 OSS 音频编程接口 308
13.5.2 ALSA 音频编程接口 310

附录：深圳优龙科技 YL2410 开发板简介 313

主要参考文献 316

第1章 Linux 驱动程序基础

Linux 是操作系统领域的奇迹。Linux 操作系统的迅猛发展，与其具有的良好特性是分不开的。Linux 是一种性能优良、源码公开、多用户、多任务操作系统，目前主要运用在大型服务器领域、网络处理应用和嵌入式系统。为了加强在嵌入式系统领域的优势，Linux 2.6 已经在内核中加入了提高中断性能和调度响应时间的改进，包括采用可抢占内核、效率更高的调度算法和同步特性。另外，Linux 2.6 内核加入了包括 S3C2410X 在内的多种微控制器的支持，并开始支持多种流行的无 MMU 单元的微控制器，如 Dragonball、ColdFire、Hitachi H8/300。掌握嵌入式 Linux 驱动开发逐渐成为一种趋势。

1.1 驱动程序的概念

驱动程序实际上就是硬件与应用程序之间的中间层。驱动程序工作在内核空间，应用程序一般运行于用户态。在内核态下，CPU 可执行任何指令，在用户态下 CPU 只能执行非特权指令。当 CPU 处于内核态，可以随意进入用户态；而当 CPU 处于用户态，只能通过特殊的方式进入内核态，比如 Linux 操作系统中的系统调用。系统调用是操作系统内核和应用程序之间的接口，设备驱动程序是操作系统内核和机器硬件之间的接口。

设备驱动程序的本质是实现逻辑设备到物理设备的转换，启动相应的 I/O 设备，发出 I/O 命令，完成相应的 I/O 操作，它是内核与外围设备数据交流的核心代码。设备驱动程序为应用程序屏蔽了硬件的细节。在应用程序看来，硬件设备只是一个设备文件， 应用程序可以像操作普通文件一样对硬件设备进行操作。

编写设备驱动必须掌握操作系统的工作原理，并对硬件设备的运行机制有清楚的认识。开发设备驱动程序必须特别注意代码的安全性。因为驱动程序的错误往往导致整个操作系统崩溃。另外，同一个设备驱动可能被不同的进程调用，所以开发设备驱动程序必须考虑并发问题的处理。

1.2 Linux 驱动程序模型

在 Linux 操作系统中，设备驱动程序对各种不同设备提供了一致的访问接口，把设备映射为一个特殊的设备文件，用户程序可以像对其他文件一样对此设备文件进行操作。Linux 支持三类硬件设备：字符设备、块设备及网络设备。

字符设备接口支持面向字符的 I/O 操作，它不经过系统的快速缓存，所以它们负责管理自己

的缓冲区结构。字符设备接口只支持顺序存取的功能，一般不能进行任意长度的 I/O 请求，而是限制 I/O 请求的长度必须是设备要求的基本块长的倍数。典型的字符设备包括鼠标、键盘、串行口等。

块设备接口主要是针对磁盘等慢速设备设计的，以免耗费过多的 CPU 等待时间。它仅支持面向块的 I/O 操作，所有 I/O 操作都通过在内核地址空间中的 I/O 缓冲区进行，它可以支持几乎任意长度和任意位置上的 I/O 请求，即提供随机存取的功能。块设备主要包括硬盘软盘设备、CD-ROM 等。

LINUX 操作系统中的网络设备是一类特殊的设备。Linux 的网络子系统主要是基于 BSD unix 的 socket 机制。在网络子系统和驱动程序之间定义有专门的数据结构（sk_buff）进行数据传递。Linux 操作系统支持对发送数据和接收数据的缓存，提供流量控制机制，提供对多协议的支持。网络接口不存在于 Linux 的文件系统中，而是在核心中用一个 device 数据结构表示。对每一个字符设备或块设备的访问是通过文件系统中相应的特殊设备文件来进行的。网络设备在做数据包发送和接收时，直接通过接口访问，不需要进行文件的操作；而对字符设备和块设备的访问都需通过文件操作界面。

Linux 系统为每个设备分配了一个主设备号与次设备号，主设备号唯一标识了设备类型，次设备号标识具体设备的实例。由同一个设备驱动控制的所有设备具有相同的主设备号。从设备号用来区分具有相同主设备号且由相同设备驱动控制的不同设备。系统中每种设备都用一种特殊的设备相关文件来表示，例如系统中第一个 IDE 硬盘表示成/dev/hda。例如串口设备/dev/tty0 与/dev/tty1，它们的主设备号为 4，次设备号分别为 0 和 1。例如主 IDE 硬盘的每个分区的从设备号都不相同。如/dev/hda2 表示主 IDE 硬盘的主设备号为 3，而从设备号为 2。块设备和字符设备的设备相关文件可以通过 mknod 命令来创建，并使用主从设备号来描述此设备。网络设备也用设备相关文件来表示，但当 Linux 寻找和初始化网络设备时才建立这种文件。在驱动程序中，可以使用下列宏获得驱动的设备号：

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

如果想把设备号转换成 dev_t 类型，使用：

```
MKDEV(int major,int minor);
```

Linux 2.6 内核的一个重要特色是提供了统一的内核设备模型。随着技术的不断进步，系统的拓扑结构越来越复杂，对智能电源管理、热插拔以及 Plug and Play 的支持要求也越来越高，Linux 2.4 内核已经难以满足这些需求。为适应这种形势的需要，Linux 2.6 内核开发了全新的设备模型，它采用 sysfs 文件系统，该文件系统是一个类似于 proc 文件系统的特殊文件系统，用于将系统中的设备组织成层次结构，并向用户模式程序提供详细的内核数据结构信息。

Linux 2.6 引入新的设备管理机制 kobject，通过这个数据结构使所有设备在底层都具有统一的接口，kobject 提供基本的对象管理，是构成 Linux 2.6 设备模型的核心结构，它与 sysfs 文件系统紧密关联，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。kobject 通常通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合。

```
struct kobject {
  char          * k_name;
  char          name[KOBJ_NAME_LEN];
  atomic_t      refcount;
  struct list_head entry;
  struct kobject   * parent;
  struct kset      * kset;
  struct kobj_type * ktype;
  struct dentry    * dentry;
};
struct kset {
  struct subsystem * subsys;
  struct kobj_type * ktype;
  struct list_head list;
  struct kobject   kobj;
  struct kset_hotplug_ops  * hotplug_ops;
};
```

在这些内核对象机制的基础上，Linux 的设备模型（/include/linux/device.h）包括设备结构 devices、驱动结构 drivers、总线结构 buses、设备类结构 classes 几个关键组件。Linux 中的任一设备在设备模型中都由一个 device 对象描述，其对应的数据结构 struct device 定义为：

```
struct device {
  struct list_head node;              //在同类设备列表中的节点
  struct list_head bus_list;          //在总线列表中节点
  struct list_head driver_list;
  struct list_head children;
  struct device     * parent;    //父设备
  struct kobject kobj;
  char  bus_id[BUS_ID_SIZE];     //父总线上的 ID
  struct bus_type   * bus;       //设备挂接的总线类型
  struct device_driver *driver;  //分配本设备的驱动
  void       *driver_data;       //私有数据
  void       *platform_data;     //平台特定数据（包括与设备相关的 ACPI，BIOS 数据）
  struct dev_pm_info power;
  u32 power_state;               //电源状态，在 ACPI 中包括 D0～D3，D0 表示全功能，
                                 //D3 表示停止
  unsigned char *saved_state;    //设备状态
  u32        detach_state;       //设备删除时进入的状态
  u64        *dma_mask;          //DMA 掩码
  u64        coherent_dma_mask;  //类似 DMA 掩码，用于 alloc_coherent mappings
  struct list_head  dma_pools;   //DMA 池
  void  (*release)(struct device * dev);
};
//注册与注销函数
int device_register(struct device * dev);
void device_unregister(struct device * dev);
```

系统中的每个驱动程序由一个 device_driver 对象描述，对应的数据结构定义为：

```
struct device driver {
    char * name;                             //设备驱动程序的名称
    struct bus_type * bus;                   //该驱动所管理的设备挂接的总线类型
    struct semaphore    unload_sem;
    struct kobject      kobj;                //内嵌 kobject 对象
    struct list_head    devices;             //该驱动所管理的设备链表头
    int  (*probe)(struct device * dev);      //指向设备探测函数，用于探测设备是否可以被
                                             //该驱动程序管理
    int  (*remove)    (struct device * dev);                //用于删除设备的函数
    void(*shutdown)   (struct device * dev);                //停止设备的函数
    int  (*suspend)   (struct device * dev, u32 state, u32 level);//挂起设备的函数
    int  (*resume)    (struct device * dev, u32 level);     //恢复设备的函数
};
//注册与注销函数
int driver_register(struct device_driver * drv);
void driver_unregister(struct device_driver * drv);
```

系统中总线由 struct bus_type 描述，定义为：

```
struct bus_type
{
    char * name;                                //总线类型的名称
    struct subsystem subsys;                    //与该总线相关的 subsystem
    struct kset drivers;                        //所有与该总线相关的驱动程序集合
    struct kset devices;                        //所有挂接在该总线上的设备集合
    struct bus_attribute * bus_attrs;           //总线属性
    struct device_attribute * dev_attrs;        //设备属性
    struct driver_attribute * drv_attrs;        //驱动程序属性
    int  (*match)(struct device * dev, struct device_driver * drv);
    int  (*hotplug)  (struct device *dev, char **envp, int num_envp, char *buffer,
    int buffer_size);
    int  (*suspend)(struct device * dev, u32 state);
    int  (*resume)(struct device * dev);
};
```

Linux 2.6 源代码中设备驱动大多数放在/drivers 目录。音频驱动则是一个例外，被放到源代码根目录下的/sound 目录中。表 1.1 为/drivers 目录下重要的子目录的介绍。

表 1.1　内核驱动目录

目录	主要内容	目录	主要内容
/drivers/char	字符型设备驱动	/drivers/media	视频采集、广播、数字电视设备
/drivers/block	块设备驱动	/drivers/base	一切驱动基本函数
/drivers/net	网络设备驱动	/drivers/usb	USB 设备驱动
/drivers/video	显示相关驱动、控制台设备、启动 LOGO	/drivers/mtd	MTD 设备驱动，包括 FLASH 驱动
/drivers/mmc	MMC/SD 卡驱动	/drivers/serial	串口设备驱动

1.3 最基本的调试手段

在内核编程中，不能使用用户态 C 库函数中的 printf()函数输出信息，而只能使用 printk()。printk 函数的用法：

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__,__LINE__);
```

宏__FILE__代表文件名，宏__LINE__代表代码在文件的行号。输出消息前的宏是优先级，内核一共有 8 个优先级，它们都有对应的宏定义。如果未指定优先级，内核会选择默认的优先级 DEFAULT_MESSAGE_LOGLEVEL。Linux 2.6 中 default_message_loglevel 是 kern_warning。如果优先级数字比 console_loglevel 变量小的话，消息就会打印到控制台上。console_loglevel 被初始化成 default_console_loglevel。注意只有在非界面环境才能看到打印输出，在图形界面下的终端里是看不到输出的。表 1.2 是 8 个日志级别。

表 1.2 错误日志级别

KERN_EMERG	紧急信息	KERN_WARNING	警告
KERN_ALERT	需要立即处理的情况	KERN_NOTICE	正常情况，但值得关注
KERN_CRIT	严重错误	**KERN_INFO**	提醒消息
KERN_ERR	硬件错误	KERN_DEBUG	调试信息

1.4 导出符号的方法

Linux2.4 内核下，默认情况时模块中的非静态全局变量及函数在模块加载后会输出到内核空间。Linux 2.6 内核下，默认情况时模块中的非静态全局变量及函数在模块加载后不会输出到内核空间，需要显式调用宏 EXPORT_SYMBOL 才能输出。所以在 Linux 2.6 内核的模块下，EXPORT_NO_SYMBOLS 宏的调用没有意义，是空操作。在同时支持 Linux 2.4 与 Linux 2.6 内核的设备驱动中，可以通过以下代码段来输出模块的内核符号。同时支持 Linux 2.4 与 Linux 2.6 的输出内核符号代码段：

```
EXPORT_NO_SYMBOLS;
EXPORT_SYMBOL(var);
EXPORT_SYMBOL(func);
```

需要注意的是如需在 Linux 2.4 内核下使用 EXPORT_SYMBOL，必须在 CFLAGS 中定义 EXPORT_SYMTAB，否则编译将会失败。从编写良好的代码风格角度出发，对于不需要输出到内核空间，而且模块中其他文件没有用到的全局变量及函数，最好显式声明为 static 类型；而需要输出的内核符号，在命名时通常要用模块名作为前缀。模块加载后，Linux 2.4 内核下可通过/proc/ksyms、Linux 2.6 内核下可通过/proc/kallsyms 查看模块输出的内核符号。

1.5 动态加载驱动程序

Linux 设备驱动属于内核的一部分，它可以采用两种方式被编译和加载：(1)直接编译进 Linux 内核，随同 Linux 启动时加载；（2）编译成一个可加载模块，使用 insmod 加载，使用 rmmod 删除。Linux 系统的可加载模块（Loadable Kernel Modules）是用于扩展 Linux 系统的功能的。使用内核模块的优点有:它们可以按照需要被动态地加载，而且不需要重新编译内核。这种方式控制了内核的大小，而模块一旦被插入内核，它就和内核其他部分一样。Linux 2.6 中的模块必须包含以下基本接口：

```
module_init(your_init_func);
module_exit(your_exit_func);
```

加载一个模块（常常只限于 root 能够使用）的命令是：

```
insmod modulename.ko
```

卸载一个内核模块的命令是：

```
rmmod modulename
```

内核代码 include/linux/module.h 中定义的宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。在装载内核模块时，用户可以向模块传递一些参数：

```
insmod modname var=value
```

如果用户未指定参数，var 将使用模块内定义的默认值。

Linux 设备驱动属于内核的一部分，它直接被编译进内核，也可以作为可加载模块动态加载。编译进内核的驱动随系统启动而加载，而作为动态加载模块则在执行 insmod 时加载。

1.6 在内核中加入新驱动

如果希望将驱动编译进内核，需要修改内核代码。下面以字符型设备为例，说明如何在 Linux 2.6 中加一个新的设备驱动。如果驱动代码文件为 pxa_smbus.c，将 pxa_smbus.c 复制到/drivers/char 目录，更改该目录下 Kconfig，增加：

```
config SMBUS_PMIC
    tristate "SMBUS Driver for PMIC"
    depends on ARCH_PXA || ARCH_SA1100
    default y
```

```
    help
```

在该目录下的 Makefile 中增添下行：

```
obj-$(CONFIG_SMBUS_PMIC)+=pxa_smbus.o
```

进入源代码目录，执行 make menuconfig 后，选择 character devices 项，进入图 1.1 所示的界面，在图中选项前如果为<*>，表示模块被编译进内核；如果为<M>，表示编译成可加载模块；如果是< >则表示不编译。如果选择<*>，用 make zImage 就可以了。如果选择<M>，则必须使用 make 命令，生成 ko 文件。

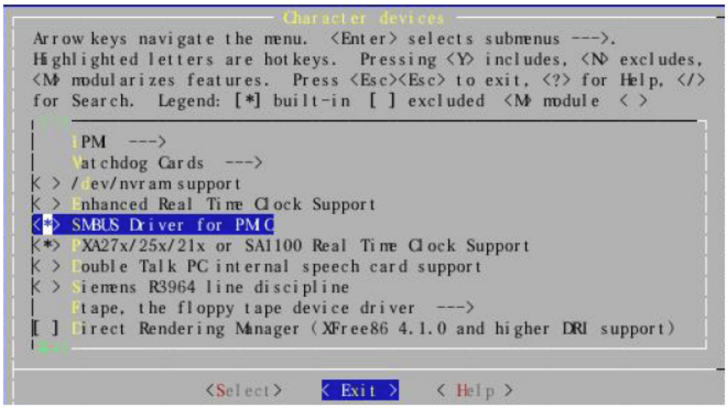


图 1.1　在内核中增加新驱动

1.7　应用程序操作接口

在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作，设备驱动需要提供应用程序操作接口，如 open 、close、read、write、seek、ioctl 等。驱动程序的原理如图 1.2 所示。

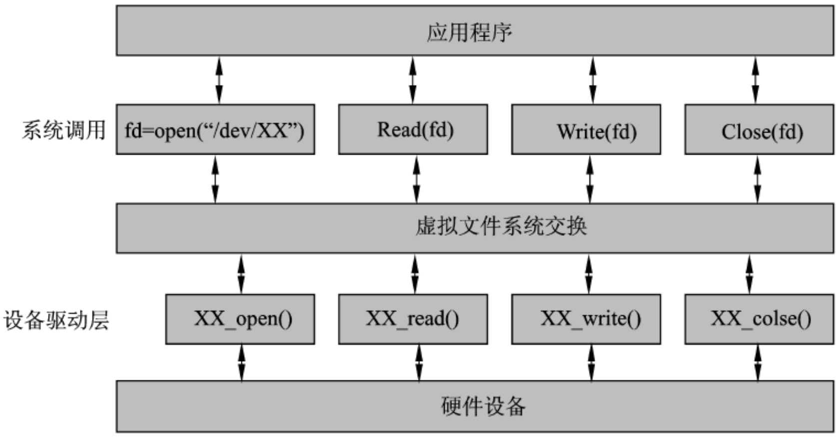


图 1.2　驱动程序的原理

内核定义了一个 struct file_operations 结构体，这个结构的每一个成员的名字都对应着一个系统调用。用户进程利用系统调用在对设备文件进行诸如读写操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数。

```
struct file_operations {
  struct module *owner;
  //模块所有者指针，一般初始化为 THIS_MODULES
  loff_t (*llseek) (struct file *, loff_t, int);
  //用来修改文件当前的读写位置，返回新位置。loff_t 为一个"长偏移量"
  ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);
  //同步读取函数。读取成功返回读取的字节数。设置为 NULL，调用时返回-EINVAL
  ssize_t (*aio_read) (struct kiocb *, char _user *, size_t, loff_t);
  //异步读取操作，为 NULL 时全部通过 read 处理
  ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);
  //同步写入函数
  ssize_t (*aio_write) (struct kiocb *, const char _user *, size_t, loff_t);
  //异步写入操作
  int (*readdir) (struct file *, void *, filldir_t);
  // 仅用于读取目录，对于设备文件，该字段为 NULL
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  //判断目前是否可以对设备进行读写操作。字段为空时，设备会被认为既可读也可写
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  // 向设备发送 I/O 控制命令的函数。不设置入口点，返回-ENOTTY
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  // 不使用 BLK 的文件系统，将使用此种函数指针代替 ioctl
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  // 在 64 位系统上，32 位的 ioctl 调用，将使用此函数指针代替
  int (*mmap) (struct file *, struct vm_area_struct *);
  // 用于请求将设备内存映射到进程地址空间。如果无此方法，将访问-ENODEV
  int (*open) (struct inode *, struct file *);
  // 打开设备的函数，如果为空，设备的打开操作永远成功，但系统不会通知驱动程序
  int (*flush) (struct file *);
  // 进程在关闭设备文件描述符副本时调用，执行未完成的操作
  int (*release) (struct inode *, struct file *);
  // file 结构释放时，将调用此指针函数，若 release 与 open 相同可设置为 NULL
  int (*fsync) (struct file *, struct dentry *, int datasync);
  // 刷新待处理的数据，如果驱动程序没有实现，fsync 调用将返回-EINVAL
  int (*aio_fsync) (struct kiocb *, int datasync);
  // 异步的 fsync 函数
  int (*fasync) (int, struct file *, int);
  // 通知设备 FASYNC 标志发生变化，如果设备不支持异步通知，该字段可以为 NULL
  int (*lock) (struct file *, int, struct file_lock *);
  // 实现文件锁，设备驱动常不去实现此 lock
  ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
  //实现进行涉及多个内存区域的单次读或写操作
  ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
  // 实现 sendfile 调用的读取部分，将数据从一个文件描述符移到另一个
```

```
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    // 实现 sendfile 调用的另一部分，内核调用将其数据发送到对应文件，每次一个数据页
    unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
    unsigned long, unsigned long);
    // 在进程地址空间找到一个合适的位置，以便将底层设备中的内存段映射到该位置
    int (*check_flags) (int);
    // 允许模块检查传递给 fcntl(F_SETEL...) 调用的标志
    int (*dir_notify) (struct file *filp, unsigned long arg);
    // 应用程序使用 fcntl 来请求目录改变通知时，调用该方法。仅对文件系统有效，驱动程序不必实现
    int (*flock) (struct file *, int, struct file_lock *);
    // 实现文件锁
};
```

特别注意，在上面的结构中，struct file 表示一个打开的文件，struct inode 表示一个磁盘上的具体文件。struct file 数据结构如下：

```
struct file {
    struct list_head f_list;
    struct dentry   *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    int f_error;
    loff_t f_pos;
    struct fown_struct f_owner;
    unsigned int f_uid, f_gid;
    struct file_ra_state f_ra;
    unsigned long f_version;
    void *f_security;
    void *private_data;
    struct address_space *f_mapping;
};
```

表 1.3 为 file 结构中重要成员的说明：

表 1.3　file 结构中与驱动相关的成员

成员	含义
f_mode	标识文件的读写权限，它通过 FMODE_READ 和 FMODE_WRITE 位来标示文件是否可读，可写
f_pos	当前读写位置，类型为 loff_t，是 64 位的数
f_flag	文件标志，主要用于进行阻塞/非阻塞型操作时检查。如 O_RDONLY，O_NONBLOCK，O_SYNC 等，驱动程序为了支持非阻塞型操作需要检查这个标志
f_op	文件操作接口函数，在驱动程序中实现
private_data	可以存放任何私有数据。一般用它指向已经分配的数据，在内核销毁 file 结构前要在 release 方法中释放内存
f_dentry	文件对应的目录项结构，一般在驱动中用 filp->f_dentry->d_inode 访问索引节点时用到它
f_vfsmnt	虚拟文件系统挂载点
f_ra	跟踪上次文件操作状态的结构指针

1.8 第一个驱动

现在就开始驱动之旅，首先实现一个简单的字符型驱动。它的功能是在内存中分配一块可读写的内存区，应用程序通过通用的文件读写接口对这块内核中的内存区域进行操作。首先定义一个字符设备以及一个 256 字节的内存块：

```
struct DEMO dev //自定义设备结构
{
    struct cdev cdev;
};
struct file_operations DEMO_fops = {
    .owner =    THIS_MODULE,
    .llseek =   DEMO_llseek,
    .read =     DEMO read,
    .write =    DEMO write,
    .ioctl =    DEMO ioctl,
    .open =     DEMO open,
    .release =  DEMO release,
};
struct DEMO_dev *DEMO_devices;
static unsigned char demo_inc=0;
static u8 demoBuffer[256];//分配的内存区
#define DEMO_MAJOR 224//主设备号
```

Linux 2.6 中模块初始化（module_init）与卸载（module_exit）是必需的。可以在模块初始化函数中注册字符设备，并在卸载函数中注销它。

```
int DEMO_init_module(void)
{
    int result;
    dev t dev = 0;
    //从设备号转换成 dev_t
    dev = MKDEV(DEMO MAJOR, DEMO MINOR);
    //注册设备号
    result = register chrdev region(dev, 1, "DEMO");
    if (result < 0)
    {
        printk(KERN_WARNING "DEMO: can't get major %d\n", DEMO_MAJOR);
        return result;
    }
    //分配 DEMO_dev，该结构是自定义的
    DEMO devices = kmalloc(sizeof(struct DEMO dev), GFP KERNEL);
    if (!DEMO_devices)
```

```
    {
        result = -ENOMEM;
        goto fail;
    }
    memset(DEMO_devices, 0, sizeof(struct DEMO_dev));
    //初始化一个字符驱动
    cdev_init(&DEMO_devices->cdev, &DEMO_fops);
    DEMO_devices->cdev.owner = THIS_MODULE;
    DEMO_devices->cdev.ops = &DEMO_fops;
    //在内核中添加字符驱动
    result = cdev_add (&DEMO_devices->cdev, dev, 1);
    if(result)
    {
        printk(KERN_NOTICE "Error %d adding DEMO\n", result);
        goto fail;
    }
    return 0;
   //失败处理
fail:
    DEMO_cleanup_module();
    return result;
}
void DEMO_cleanup_module(void)
{
    dev_t devno = MKDEV(DEMO_MAJOR, DEMO_MINOR);
    //删除驱动
    if (DEMO_devices)
    {
        cdev_del(&DEMO_devices->cdev);
        kfree(DEMO_devices);
    }
    //释放设备号
    unregister_chrdev_region(devno,1);
}
module_init(DEMO_init_module);
module_exit(DEMO_cleanup_module);
```

下面需要实现驱动对应的文件操作。第一个操作接口就是 open：

```
int DEMO_open(struct inode *inode, struct file *filp)
{
    struct DEMO_dev *dev;
    //防止多次打开设备
    if(demo_inc>0)return -ERESTARTSYS;
    demo_inc++;
    //获取 DEMO_dev 结构，并将它赋给 filp->private_data，以后可以用 filp->private_data
```

```
    //访问设备 DEMO_dev 结构
    dev = container_of(inode->i_cdev, struct DEMO_dev, cdev);
    filp->private_data = dev;
    return 0;
}
```

container_of 宏定义如下，它的作用是：设 ptr 是某个 type 类型的结构中的 member 成员变量的地址，该宏的结果是得到该 type 结构的地址。

```
#define container_of(ptr, type, member) ({
    const typeof( ((type *)0)->member ) *_mptr = (ptr);
    (type *)( (char *)_mptr - offsetof(type,member) );})
```

会发现在 read，write，seek 等函数的参数中没有出现 struct inode *inode。在这些函数的实现中调用 filp->private_data 可以获得驱动的信息。与 open 相对的是：

```
int DEMO_release(struct inode *inode, struct file *filp)
{
    //引用计数
    demo inc--;
    return 0;
}
```

打开设备后，如果要对设备进行读写操作，就必须在驱动中提供相应的接口。参数 filp 是文件指针，buf 是用户空间的数据缓冲区，count 是请求传输的字节数，f_pos 是文件当前的偏移量。

```
ssize_t DEMO_read(struct file *filp, char _user *buf, size_t count,loff_t *f_pos)
{
    int result;
    //得到文件的当前位置
    loff_t pos= *f_pos;
    //判断位置的合法性
    if(pos>=256)
    {
        result=0;
        goto out;
    }
    if(count>(256-pos))
    {
        count=256-pos;
    }
    pos += count;
    //把数据写到应用程序空间
    if (copy_to_user(buf,demoBuffer+*f_pos,count))
    {
      count=-EFAULT;
```

```
        goto out;
    }
//改变文件的当前位置
    *f_pos = pos;
out:
    return count;
}
```

写操作的参数与读操作的参数相同。

```
ssize_t DEMO_write(struct file *filp, const char _user *buf, size_t count,loff_t
*f_pos)
{
    ssize_t retval = -ENOMEM;
    loff_t pos= *f_pos;
    //判断位置的合法性
    if(pos>=256)
    {
        goto out;
    }
    if(count>(256-pos))
    {
        count=256-pos;
    }
    pos += count;
    //将数据复制到用户空间
    if (copy_from_user(demoBuffer+*f_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    //改变文件的当前位置
    *f_pos = pos;
    return count;
out:
    return retval;
}
```

设备要支持 seek，必须在驱动中实现 seek 操作。参数 off 表示偏移量，whence 告诉驱动从何处开始移动 off 字节。

```
loff_t DEMO_llseek(struct file *filp, loff_t off, int whence)
{
    loff_t pos;
    pos = filp->f_pos;
    //根据开始位置（0，当前）来调整文件指针
    switch (whence)
```

```
        {
        case 0://从 0 开始
            pos = off;
            break;
        case 1://从当前位置开始
            pos += off;
            break;
        case 2:
        default:
            return -EINVAL;
        }
        //超出范围
        if ((pos>256) || (pos<0))
        {
            return -EINVAL;
        }
        return filp->f_pos=pos;
    }
```

通过 Ioctl 可以在应用层对设备进行一些参数修改。DEMO_ioctl 函数中 unsigned int cmd 就是命令号，arg 是命令参数。

```
    int DEMO_ioctl(struct inode *inode, struct file *filp,unsigned int cmd, unsigned
    long arg)
    {
          if(cmd==COMMAND1)
          {
               printk("ioctl command1 successfully\n");
               return 0;
          }
          if(cmd==COMMAND2)
          {
               printk("ioctl command2 successfully\n");
               return 0;
          }
          printk("ioctl error\n");
       return -EFAULT;
    }
```

安装交叉编译环境非常简单，下载 cross-3.3.2.tar.bz2，解压后复制到/usr/local/arm 目录下，再通过在/etc/bashrc 文件中添加下面的命令将该路径加入到环境路径中就可以了。

```
    export PATH=/usr/local/arm/3.3.2/bin:$PATH
```

驱动代码完成后，需要编写一个 makefile：

```
AR= ar
ARCH = arm
CC = arm-linux-gcc
CFLAGS += $(DEBFLAGS) -Wall
CFLAGS += -I$(LDDINC)
LDFLAGS = -Xlinker -rpath-link /embedded/tools/usr/lib/gcc/arm-linux/4.0.0
ifneq ($(KERNELRELEASE),)
# call from kernel build system
obj-m := demo.o
else
KERNELDIR ?= /home/s3c2410/linux-2.6.9
PWD       := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) LDDINC=$(PWD)/../include modules
endif
```

执行 make 命令后，生成 demo.ko。启动开发板后，在 minicom 终端下进入开发板/tmp 目录，执行 rz 命令，将 demo.ko 发送到目标板上。随后执行动态加载命令：

```
#insmod demo.ko
```

这时可以用命令 lsmod 查看动态加载模块：

```
#lsmod
```

当然，可以用 cat /proc/modules 命令查看加载的模块，用 cat /proc/devices 命令查看设备名等信息。如果要卸载该模块，输入命令：

```
#rmmod demo
```

使用 mknod 建立设备节点，这样应用程序就可以通过该节点访问设备。

```
#mknod /dev/fgj c 224 0
```

现在编写一个应用程序来测试驱动：

```
main()
{
  int fd;
  int i;
 char data[256];
  int retval;
  fd=open("/dev/fgj",O RDWR);
  if(fd==-1)
  {
     perror("error open\n");
     exit(-1);
  }
```

```
    printf("open /dev/smbus successfully\n");
    //发送 COMMAND1 命令
    retval=ioctl(fd,COMMAND1,0);
    if(retval==-1)
    {
      perror("ioctl error\n");
      exit(-1);
    }
    printf("send command1 successfully\n");
    //对设备进行写操作
    retval=write(fd,"fgj",3);
    if(retval==-1)
    {
      perror("write error\n");
      exit(-1);
    }
    //寻找文件起始位置
    retval=lseek(fd,0,0);
    if(retval==-1)
    {
      perror("lseek error\n");
      exit(-1);
    }
    //对设备进行读操作
    retval=read(fd,data,3);
   if(retval==-1)
    {
      perror("read error\n");
      exit(-1);
    }
     printf("read successfully:%s\n",data);
    close(fd);
  }
```

第2章 Linux 驱动开发核心技术

Linux 内核非常复杂，主要由进程调度、内存管理、虚拟文件系统、网络子系统、进程通信、设备管理等几个子系统组成，各个子系统之间相互依赖。嵌入式系统开发人员重点应该学习与驱动和应用相关的内容。本章将介绍 Linux 内核中驱动开发相关的接口，包括同步机制、时间、内存映射和中断处理等。

2.1　同步机制

中断处理、多任务环境、多处理器（SMP）是现代操作系统的特征。当多个进程、线程同时访问同一个资源时，可能导致错误。内核需要提供并发控制机制，对公共资源的访问进行同步控制，确保共享资源的安全访问。Linux 操作系统中包含众多的同步机制，包括信号量（semaphore）、自旋锁（spinlock）、原子操作（atomic operation）、读写锁（rwlock）、RCU（包含在 Linux 2.6 内核中）和 seqlock（包含在 Linux 2.6 内核中），每种机制应用在不同场合。

2.1.1　自旋锁

自旋锁（spinlock）主要用在 SMP 的环境下。在单处理器环境中，spin_lock 和 spin_unlock 的作用仅限于禁止和允许内核抢占。自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元占有，调用者就一直循环查看是否该自旋锁的保持者已经释放了锁。这种特性避免了调用进程的挂起，用自旋来取代进程切换。与自旋锁相关的函数主要有：

```
spinlock_t spin;
//定义自旋锁
spin_lock_init(lock);
//初始化自旋锁
spin_lock(lock);
//获得自旋锁，如果获得锁将立即返回，否则在原地等待，直到获得锁。在任何时刻，最多只能有一个保持者
spin_trylock(lock);
//尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假
spin_unlock(lock);
//释放自旋锁
spin_lock_irq();
spin_lock_irqsave(lock, flags) ;
```

```
//中断安全版本的自旋锁，获得自旋锁的同时禁止本地 CPU 上的中断。当自旋锁可以被中断上下文获得时，
//必须使用这个函数
spin_lock_bh(lock);
//软件中断安全版本的自旋锁
```

2.1.2　信号量

信号量与自旋锁功能很相似，但它们实现机理完全不一样。信号量的调用会引起调用者睡眠，除非获得锁。信号量适用于单处理器系统或多处理器系统中。信号量允许并行访问，即可以有多个内核控制路径同时掌握该信号量，它所允许的并行访问数目是在信号量创建的时候定义的。互斥对象（mutex）是一种特殊的信号量，它所保护的资源在同一时刻只允许一个内核控制路径访问。与信号量相关的函数主要有：

```
struct semaphore sem;
//定义信号量
void sema_init (struct semaphore *sem, int val);
//初始化信号量
void init_MUTEX (struct semaphore *sem);
//等同于 sema_init (struct semaphore *sem, 1);
void init_MUTEX_LOCKED (struct semaphore *sem);
//等同于 sema_init (struct semaphore *sem, 0);
void down(struct semaphore * sem);
//获得信号量，它会导致睡眠，因此不能在中断上下文使用
int down_interruptible(struct semaphore * sem);
//获得信号量，能被信号打断；该函数用返回值来区分是正常返回还是被信号中断，如果返回 0，表示获得
//信号量正常返回，如果被信号打断，返回-EINTR
int down_trylock(struct semaphore * sem);
//尝试获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导
//致调用者睡眠，可以在中断上下文使用
void up(struct semaphore * sem);
//释放信号量，唤醒等待者
```

2.1.3　原子操作

原子操作是一系列不可中断的操作的集合，它的执行过程是封闭的，不可打断。“原子”实际是使用了物理学里的物质微粒的概念。原子操作的使用与其他锁同步机制的差异很大，可以将要保护的资源定义成原子型整数 atomic_t 类型，然后调用原子操作的接口对这些资源进行修改。对整数进行原子操作的接口包括：

```
typedef struct { volatile int counter; } atomic_t;
//原子类型定义
atomic_read(atomic_t * v);
//对原子类型的变量进行原子读操作，它返回原子类型的变量 v 的值
```

```
atomic_set(atomic_t * v, int i);
//该函数设置原子类型的变量 v 的值为 i
void atomic_add(int i, atomic_t *v);
//该函数给原子类型的变量 v 增加值 i
atomic_sub(int i, atomic_t *v);
//该函数从原子类型的变量 v 中减去 i
int atomic_sub_and_test(int i, atomic_t *v);
//从原子类型的变量 v 中减去 i，并判断结果是否为 0，如果为 0，返回真，否则返回假
int atomic_dec_and_test(atomic_t *v);
//对原子类型的变量 v 减 1，并判断结果是否为 0，如果为 0，返回真，否则返回假
void atomic_inc(atomic_t *v);
//该函数对原子类型变量 v 原子地增加 1
void atomic_dec(atomic_t *v);
//该函数对原子类型的变量 v 原子地减 1
```

对位进行原子操作的函数包括：

```
int set_bit(int nr, void *addr);
//对给定地址 addr 的第 nr bit 进行置位
int clear_bit(int nr, void *addr);
//对给定地址 addr 的第 nr bit 进行清位
int test_bit(int nr，void *add);
//检测给定地址 addr 的第 nr bit 的值
int change_bit(int bit, void *addr);
//修改给定地址 addr 的第 nr bit
int test_and_set_bit(int nr, void *addr);
//设置给定地址 addr 的给定 bit 并返回它以前的值
int test_and_clear_bit(int nr, void *addr);
//清除给定地址 addr 的给定 bit 并返回它以前的值
int test_and_change_bit(int nr, void *addr);
//修改给定地址 addr 的给定 bit 并返回它以前的值
atomic_clear_mask(mask, addr);
//清除所有 mask 指定的位
atomic_set_mask(mask, addr);
//设置所有 mask 指定的位
```

2.1.4 读写锁（rwlock）

读写锁实际是一种特殊的自旋锁，它把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。读写锁的函数看上去与自旋锁很相似，只是读者和写者需要不同的获得和释放锁的函数。这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU 数。写者是具有排它性的，一个读写锁同时只能有一个写者或多个读者（与 CPU 数相关），但不能同时既有读者又有写者。在读写锁保持期间也是抢占失效的。如果读写锁当前没有读者，

也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

初始化读写锁的方法有两种：

```
rwlock_t x;
rwlock_init(x) ;
该宏用于动态初始化读写锁 x。一般情况下等同于:
rwlock_t x = RW_LOCK_UNLOCKED
```

下面一组函数是最基本的读写锁的函数：

```
read lock(lock);
//读者可以使用该宏来获得读写锁，如果不能获得锁，它将自旋，直到获得该读写锁
read unlock(lock);
//读者使用该宏来释放读写锁 lock
write_lock(lock);
//写者可以使用该宏来获得读写锁，如果不能获得锁，它将自旋，直到获得该读写锁
write_unlock(lock);
//写者使用该宏来释放读写锁 lock
```

相对于自旋锁中的 spin_trylock(lock)，读写锁中分别为读写提供了尝试获取锁，并立即返回函数：

```
read_trylock(lock)
//读者用它来尝试获得读写锁 lock，如果能够立即获得读写锁，它就获得锁并返回真，否则如果不能获得
//锁，返回假
write trylock(lock)
//写者用它来尝试获得读写锁 lock，如果能够立即获得读写锁，它就获得锁并返回真，否则如果不能获得
//锁，返回假
```

硬中断安全的读写锁函数包括：

```
read_lock_irq(lock)
//用于读者获得读写锁，并禁止本地中断
read_unlock_irq(lock)
//用于读者释放读写锁，并使能本地中断
write_lock_irq(lock)
//用于写者获得读写锁，并禁止本地中断
write unlock irq(lock)
//用于写者释放读写锁，并使能本地中断
read lock irqsave(lock, flags)
//用于读者获得读写锁，同时保存中断标志，并禁止本地中断
read unlock irqrestore(lock, flags)
//用于读者释放读写锁，同时恢复中断标志，并使能本地中断
write_lock_irqsave(lock, flags)
```

```
//用于写者获得读写锁，同时保存中断标志，并禁止本地中断
write_unlock_irqrestore(lock, flags)
//用于写者释放读写锁，同时恢复中断标志，并使能本地中断
```

下面是软中断安全的读写锁函数：

```
read_lock_bh(lock)
//用于读者获得读写锁，并禁止本地软中断
read_unlock_bh(lock)
//用于读者释放读写锁，并使能本地软中断
write_lock_bh(lock)
//用于写者获得读写锁，并禁止本地软中断
write_unlock_bh(lock)
//用于写者释放读写锁，并使能本地软中断
```

2.1.5 seqlock 机制

顺序锁（seqlock）是一种免锁机制，可提供共享资源的快速访问。新增的 seqlock 主要用于：（1）少量的数据保护；（2）数据比较简单（没有指针），并且使用频率很高；（3）写入访问很少发生，并且快速地执行；（4）写操作比读操作的优先级高。seqlock 是一种写优先锁，如果想对它加写锁，只要此时没有别的写锁，那么不管有没有、有多少读锁加在它上面，都会加锁成功。读者进入临界区，不需要关闭内核抢占，而写者进入临界区，会自动关闭内核抢占。下面是 seqlock_t 结构，其中包含一个自旋锁和一个顺序变量 sequence，当 sequence 为奇数时，说明没有写者，否则说明有写者。

```
typedef struct {
    unsigned sequence;
    spinlock_t lock;
} seqlock_t;
```

顺序锁的函数与 rwlock 非常相似，这里就不一一注释了。

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;
seqlock_t lock2; seqlock_init(&lock2);
//初始化 seqlock
int write_tryseqlock(seqlock_t *sl);
void write_seqlock(seqlock_t *sl);
void write_sequnlock(seqlock_t *sl);
void write_seqlock_irqsave(seqlock_t *sl, long flags);
void write_sequnlock_irqrestore(seqlock_t *sl, long flags);
void write_seqlock_irq(seqlock_t *sl);
void write_sequnlock_irq(seqlock_t *sl);
void write_seqlock_bh(seqlock_t *sl);
void write_sequnlock_bh(seqlock_t *sl);
//以上为写者函数，以下与读者有关
```

```
unsigned int read_seqbegin(seqlock_t *sl);
//获取 sequence
int read_seqretry(seqlock_t *sl, unsigned int iv);
//在两种情况下返回 1: （1）当 sequence 为奇数时; （2）当 sequence 发生变化时
unsigned int read_seqbegin_irqsave(seqlock_t *sl, long flags);
int read_seqretry_irqrestore(seqlock_t *sl, unsigned int iv, long flags);
```

使用 seqlock 进行写操作的流程：

```
write_seqlock(seqlock);// 获取锁并增加计数
//对共享数据进行写访问
write_sequnlock(seqlock);// 释放锁并增加计数
```

使用 seqlock 进行读操作的流程：

```
do {
seq= read_seqbegin(seqlock);
    //对共享数据进行读访问
} while (read_seqretry(seqlock, seq);
```

2.1.6 RCU

RCU（Read-Copy Update）是一种高性能的锁机制，具有很好的扩展性，但是这种锁机制的使用范围比较窄，它只适用于读多写少的情况。顾名思义，RCU 就是读-备份修改，它是基于其原理命名的。对于被 RCU 保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先备份一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作时。

与 RCU 相关的读者函数包括：

```
#define rcu_read_lock()     preempt_disable()
//进入读端临界区标记
#define rcu read unlock()   preempt enable()
//退出读端临界区
```

与 RCU 相关的写者函数包括：

```
struct rcu_head {
    struct rcu_head *next;//下一个 rcu_head
    void (*func)(struct rcu_head *head);//获得竞争条件后的处理函数
};
synchronize_rcu(void);
//在进程上下文使用。它将阻塞写者，直到所有的读者已经完成读端临界区，写者才可以继续下一步操作
synchronize sched();
//在进程上下文使用。等待所有 CPU 都处在可抢占状态，保证所有中断（不包括软中断）处理完毕
```

```
void call_rcu(struct rcu_head *head,void (*func)(void *arg),void *arg);
//它不会使写者阻塞，可以在中断上下文或软中断中使用。函数 synchronize_rcu 的实现实际上使用
//函数 call_rcu
```

使用 RCU 的读操作流程如下：

```
int a;//目标资源
rcu read lock();
//read a;
rcu_read_unlock();
```

使用 synchronize_rcu 的写操作流程如下：

```
DEFINE_SPINLOCK(foo_spinlock);
Int a_new;
spin_lock(&foo_spinlock);
//a_new=a;
//write a_new;
synchronize_rcu();
//a=a new;
spin_unlock(&foo_spinlock);
```

使用 call_rcu 的写操作流程如下：

```
Struct protectRcu
{
    int protect;
    struct rcu_head rcu;
};
struct protectRcu*global_pr;
//一般用来释放老的数据
void callback_function(struct rcu_head *r)
{
    struct protectRcu *t;
    t = container_of (r, struct protectRcu, rcu );
    kfree (t);
}
void write_process ()
{
    struct protectRcu *t, *old;
    t = kmalloc (sizeof (*t), GFP_KERNEL );//创建副本
    spin_lock (&foo_spinlock);
    t-> protect= xx;
    old= global_pr;
    global_pr=t;//用副本替换
    spin unlock (&foo spinlock);
    call rcu (old->rcu , callback function);
}
```

表 2.1 是 RCU 与 rwlock、spinlock 的函数比较：

表 2.1　RCU 与 rwlock、spinlock 的比较

同步机制	spinlock	rw lock	RCU
函数比较	spinlock_t	relock_t	spinlock_t
	spin_lock()	read_lock()	rcu_read_lock()
	spin_unlock()	read_unlock()	rcu_read_unlock()
	spin_lock()	write_lock()	spin_lock()
	spin_unlock()	write_unlock()	spin_unlock()

最后，总结一下以上各种同步机制的应用场合如表 2.2 所示：

表 2.2　各种同步机制的比较

类型	机制	应用场合
spinlock	使用忙等方法，进程不挂起	（1）用于多处理器间共享数据 （2）在可抢占的内核线程里的共享数据 （3）自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用，例如中断上下文
信号量	阻塞式等待，进程挂起	（1）适合于共享区保持时间较长的情况 （2）只能用于进程上下文
原子操作	数据的原子访问	（1）共享的简单数据类型：整型、比特型 （2）适合高效率的场合
rwlock	特殊的自旋锁	（1）允许同时读共享资源，但只能有一个写 （2）读优先于写，读写不能同时
顺序锁	一种免锁机制，基于访问计数	（1）允许同时读共享资源，但只能有一个写 （2）写优先于读，读写可以同时
RCU	通过副本的免锁访问	（1）对读占主要的场合提供高性能 （2）读访问不必获取锁、不必执行原子操作或禁止中断
关闭中断	通过禁止中断的手段，排除单处理器上的并发，会导致中断延迟	（1）中断与正常进程共享数据 （2）多个中断共享数据 （3）临界区一般很短

2.2　完成事件

2.1 节的同步机制都是为实现资源保护而提出的。完成事件本质上也是一种简单的同步机制，它适合于一些需要睡眠和唤醒的情形。如果要在任务中实现简单睡眠直到其他进程完成某些处理过程为止，完成事件无疑是最容易的，并且不会引起资源竞争。完成事件使用下面的结构描述：

```
struct completion {
    unsigned int done; //标志是否已经做完，未做完就是负值，代表等待的个数；完成为 0
    wait queue head t wait;
};
```

完成事件可以动态或静态地初始化：

```
DECLARE COMPLETION(my comp);
//静态地初始化宏
void init_completion(struct completion *x);
//动态地初始化一个完成事件
```

等待完成事件的函数如下：

```
wait_for_completion(struct completion *x);
等待一个完成事件，并使 x->done--
wait_for_completion_interruptible(struct completion *x);
可中断的 wait_for_completion
unsigned long wait_for_completion_timeout(struct completion *x, unsigned long
timeout);
带超时处理的 wait_for_completion
```

唤醒等待进程的函数包括 complete 和 complete_all。complete 函数在 wait_for_completion 之前调用，完成事件仍然能够正常工作。

```
complete(struct completion *x);
唤醒一个 x 的等待进程，使 x->done++
complete_all(struct completion *)
唤醒所有等待的进程
```

2.3　阻塞与非阻塞

阻塞操作是指在执行 I/O 操作时，若不能获得资源，则进程挂起直到满足可操作的条件时再进行操作。非阻塞操作是指在执行 I/O 操作时，如果设备没有准备好，并不挂起，立即返回。被挂起的进程进入睡眠状态，直到等待的条件被满足。显式的非阻塞 I/O 由 flip->f_flags 中的 O_NONBLOCK 标志决定。

```
Fcntl(fd,F_SETFL,fcntl(fd,F_GETFL)|O_NONBLOCK);
```

可以使用等待队列来实现阻塞式访问：

```
void init_waitqueue_head(wait_queue_head_t *q);
//初始化一个等待队列头
wait_event(wq, condition)
//不可中断的等待
wait_event_interruptible(wq, condition)
//可中断的等待
wait event timeout(wq, condition, timeout)
//超时返回
wait_event_interruptible_timeout(wq, condition, timeout)
```

```
//可中断并超时返回
wake_up(wait_queue_head_t *q)
//唤醒所有等待 q 的进程
wake_up_interruptible(wait_queue_head_t *q)
//只唤醒执行可中断休眠的进程
```


还有一个与设备阻塞与非阻塞访问息息相关的论题，即 select 和 poll。poll 和 select 用于查询设备的状态，以便用户程序获知是否能对设备进行非阻塞地访问，它们都需要设备驱动程序中的 poll 函数支持。一个典型的 poll 函数的实现（/driver/media/video/meye.c）如下：

```
static unsigned int meye_poll(struct file *file, poll_table *wait)
{
    unsigned int res = 0;
    mutex_lock(&meye.lock);
    poll_wait(file, &meye.proc_list, wait);
    if (kfifo_len(meye.doneq))//数据已经准备就绪
        res = POLLIN | POLLRDNORM;
    mutex_unlock(&meye.lock);
    return res;
}
```

驱动程序中 poll 函数中最主要用到的一个函数是 poll_wait，其原型如下：

```
void poll_wait(struct file *filp, wait_queue_heat_t *queue, poll_table * wait);
```

poll_wait 函数所做的工作是把当前进程添加到 wait 参数指定的等待列表（poll_table）中。poll_wait 函数本身并不阻塞，真正的阻塞动作是在上层的 do_select/do_poll 系统调用中完成的。do_select/do_poll 系统调用就是调用驱动程序中的 poll 函数判断设备是否就绪。驱动程序中的 poll 函数返回的标志如下：

```
POLLIN: 设备可以无阻塞地读取。
POLLRDNORM: 数据已经就绪。
POLLRDBAND:可以从设备读带外数据，仅用于 DECnet 代码中。
POLLPRI: 可以无阻塞地读取带外数据。
POLLHUP: 当前读取设备的进程到达文件尾部。
POLLERR: 设备发生错误。
POLLOUT: 设备可以无阻塞地写入。
POLLWRNORM: 同上。
POLLWRBAND: 类似 POLLRDBAND，描述带外数据的可以被写入设备。
```

程序中要用到 BSD UNIX 中引入的 select 函数，其原型为：

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select()监视的读、写和异常处理的文件描述符集

合，numfds 的值是需要检查的号码最高的文件描述符加 1。timeout 参数是一个指向 struct timeval 类型的指针，它可以使 select()在等待 timeout 时间后若没有文件描述符准备好则返回。select 函数通常与下列函数同时使用：

```
FD_ZERO(fd_set *set)//清除一个文件描述符集
FD_SET(int fd,fd_set *set)//将文件描述符 fd 加入文件描述符集中
FD_CLR(int fd,fd_set *set)//将文件描述符 fd 从文件描述符集中清除
FD_ISSET(int fd,fd_set *set)//判断文件描述符 fd 是否被置位
```

2.4 时间

2.4.1 Linux 下延迟

Linux 内核代码中有一个全局变量 jiffies，这是一个 32 位的无符号整数，用来表示自内核上一次启动以来的时钟滴答次数。每发生一次时钟滴答，内核的时钟中断处理函数 timer_interrupt()会将该全局变量 jiffies 加 1。

```
extern unsigned long volatile jiffies;
```

几个关于时间比较的宏：

```
#define time_after(a,b) //a 是否在 b 之后
#define time_before(a,b) //a 是否在 b 之前
#define time_after_eq(a,b)// a 是否在 b 之后或等于 b
#define time_before_eq(a,b) // a 是否在 b 之前或等于 b
```

可以调用上面的宏实现长延迟：

```
while(time_after(curjffies,j1));//如果 jiffies 大于 j1
```

这种延迟是一种忙等延迟，非常影响系统的效率。最佳的长延迟方法是让核心代劳：

```
long sleep_on_timeout(wait_queue_head_t *q, long timeout);
long interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout);
```

短延迟一般使用下面的函数实现：

```
#define ndelay(n)   //纳秒级延迟
#define udelay(n)   //微秒级延迟
#define mdelay(n)  //毫秒级延迟
```

2.4.2 内核定时器

Linux 内核中定义了一个 timer_list 结构，利用它可以实现内核定时器功能：

```
struct timer list {
  struct list head list;
  unsigned long expires; //定时器到期时间
  unsigned long data; //作为参数被传入定时器处理函数
  void (*function)(unsigned long);//回调处理函数
};
```

与定时器相关的函数包括：

```
void add_timer(struct timer_list * timer);
//增加定时器
int del_timer(struct timer_list * timer);
//删除未到期的定时器。到期的定时器会自动删除
int mod_timer(struct timer_list *timer, unsigned long expires);
//修改定时器的 expire 值
```

2.5　内存分配与映射

Linux 操作系统下包括以下类型的地址类型，如表 2.3 所示：

表 2.3　linux 中的各种地址

类型	说明
用户虚拟地址	用户空间程序的地址
物理地址	处理器与系统内存之间使用的地址
总线地址	外围总线与内存之间使用的地址
内核逻辑地址	内存的部分或全部映射，大多数情况下，它与相关联的物理地址仅差一个偏移量，两者的映射是线形的，如 Kmalloc 分配的内存
内核虚拟地址	内核空间的地址映射到物理地址上，但映射不必是线形的。所有的逻辑地址都是内核虚拟地址，如 vmalloc 分配的内存

2.5.1　内存分配与释放

在 Linux 内核模式下，不能使用用户态的 malloc()和 free()函数申请和释放内存。进行内核编程时，最常用的内存申请和释放函数为在 include/linux/kernel.h 文件中声明的 kmalloc()和 kfree()，其原型为：

```
void *kmalloc(unsigned int len, int priority);
void kfree(void *_ptr);
```

kmalloc 的 priority 参数通常设置为 GFP_KERNEL，如果在中断服务程序里申请内存，则要用 GFP_ATOMIC 参数，因为使用 GFP_KERNEL 参数可能会引起睡眠，不能用于非进程上下文

中（在中断中是不允许睡眠的）。kmalloc 一般用来分配小于 128KB 的内存。如果要分配大块的内存，应使用面向页的技术。分配页面的函数：

```
unsigned long get_zeroed_page(unsigned int gfp_mask);
//返回一个单个的，零填充的页
unsigned long _get_free_pages(unsigned int gfp_mask, unsigned int order);
//直接获取整页的内存（页数是 2 的幂）
#define free_page(addr)
free_pages((addr),0);
void free_pages(unsigned long addr, unsigned int order);
//释放面向页分配的函数
```

另外一种分配方法是 vmalloc。这个函数分配一片连续的虚拟内存。也就是说返回的虚拟内存虽然是连续的，但是映射到的物理内存是不连续的，而且可能与物理地址是一一对应的（不同于 kmalloc 和_get_free_pages）。在使用分配到的内存时，页表的查询比较频繁，所以效率相对较低。

```
void *vmalloc(unsigned long size);
void vfree(void *addr);
```

2.5.2　用户态和内核态内存交互

由于内核态和用户态使用不同的内存定义，所以二者相互不能直接访问对方的内存。而应该使用 Linux 中的用户和内核态内存交互函数（这些函数在 include/asm/uaccess.h 中被声明）：

```
int access_ok(int type, const void *addr, unsigned long size);
//当一个指针指向用户空间时，必须确保指向的用户地址是合法的，而且对应的页面也已经映射，可以采
//用 access_ok 检测，其中 type 有两个选项：VERIFY_READ、VERIFY_WRITE，对应于内存读写
unsigned long copy_from_user(void *to, const void *from, unsigned long n);
unsigned long copy_to_user (void * to, void * from, unsigned long len);
//函数返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0
int  put_user(dataum,ptr);
int  get_user(local,ptr);
//内核空间和用户空间的单值交互（如 char、int、long）
```

在访问用户空间的内存时，可以使用下面的方法先检查用户空间的指针是否合法：

```
char kernelbuffer[100];
static ssize_t demo_read(struct file *file, char _user *buffer, size_t count, loff_t
*ppos)
{
    if (!access_ok(VERIFY_WRITE, buffer, count))
        return -EFAULT;
    if (copy_to_user(buffer, kernelbuffer, count))
            return -EFAULT;
    return count;
}
```

2.5.3 内存池

在 Linux 2.5 的开发过程中，加入了内存池的概念，以满足无间断内存分配。其思想是预分配一个内存池，并保留到真正需要的时候。内存池可以用来分配整页的内存或小内存，它所处理的内存单元通常称为内存元素（memory element）。内存池只能被它的拥有者使用，内存池拥有者通常情况下不使用内存池，只有在动态内存不足时才使用。内存池用 mempool_t 来描述：

```
typedef struct mempool_s {
    spinlock_t lock;//竞态保护
    int min_nr;//内存池中元素的最大数量
    int curr_nr;//当前的元素数目
    void **elements;//保留元素的指针
    void *pool_data;//拥有者的私有数据
    mempool_alloc_t *alloc;//分配一个元素的方法
    mempool_free_t *free;//释放一个元素的方法
    wait_queue_head_t wait;//等待队列
} mempool_t;
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
//pool_data 是分配和回收函数用到的指针，gfp_mask 是分配标记。只有当_GFP_WAIT 标记被指定时，
//分配函数才会休眠
```

与内存池相关的函数在头文件 linux/mempool.h 中，主要包括：

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,mempool_free_t
*free_fn, void *pool_data);
//创建一个内存池，min_nr 是需要预分配对象的数目，alloc_fn 和 free_fn 是指向内存池机制提供
//的标准对象分配和回收例程的指针
void *mempool_alloc(mempool_t *pool, int gfp_mask);
//在内存池中分配元素
void mempool_free(void *element, mempool_t *pool);
//在内存池中回收元素
void mempool_destroy(mempool_t *pool);
//销毁内存池
```

除了为内存分配引入了内存池之外，Linux 2.5 内核还引入了三个用于常规内存分配的新的 GFP 标记（/include/gpf.h），它们是：

```
_GFP_REPEAT: 告诉页分配器尽力去分配内存。如果内存分配失败过多，应该减少这个标记的使用。
_GFP_NOFAIL: 不能出现内存分配失败。这样，由于调用者被转入休眠状态，可能需要一段比较长的时间
才能完成分配，调用者的需求才能得到满足。
_GFP_NORETRY: 保证分配失败后不再重试，而向调用者报告失败状态。
```

2.5.4 物理地址到虚拟地址的映射

CPU 对外设 I/O 端口物理地址的编址方式有两种：一种是 I/O 映射方式（I/O-mapped），另一种是内存映射方式（Memory-mapped）。具体采用哪一种方式取决于 CPU 的体系结构。在 x86 体系中，为外设专门实现了与 RAM 内存地址不同的一个单独的地址空间，也就是 I/O 映射方式。可以使用 outb、inb 等函数直接操作这些 I/O 端口。而在 PowerPC、m68k 和 ARM 等体系中，外设 I/O 端口具有与内存相同的物理地址，这样的方式是内存映射方式。在内存映射方式下，外设的 I/O 内存资源的物理地址是已知的，由硬件的设计决定。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问 I/O 内存资源，而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些 I/O 内存资源。Linux 在 io.h 头文件中声明了函数 ioremap()，用来将 I/O 内存资源的物理地址映射到核心虚地址空间（3GB-4GB）中，原型如下：

```
void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags);
//物理地址映射到核心虚地址空间
void iounmap(void * addr);
//取消 ioremap 所做的地址映射
```

这两个函数都实现在 mm/ioremap.c 文件中。

在将 I/O 内存资源的物理地址映射成核心虚地址后，理论上讲就可以像读写 RAM 那样直接读写 I/O 内存资源了。为了保证驱动程序跨平台的可移植性，应该使用 Linux 中特定的函数来访问 I/O 内存资源，而不应该通过指向核心虚地址的指针来访问。如在 ARM 平台上，读写 I/O 的函数如下所示：

```
#define _raw_writeb(v,a) (_chk_io_ptr(a), *(volatile unsigned char _force *)(a) = (v))
#define _raw_writew(v,a) (_chk_io_ptr(a), *(volatile unsigned short _force *)(a) = (v))
#define _raw_writel(v,a) (_chk_io_ptr(a), *(volatile unsigned int _force *)(a) = (v))
#define _raw_readb(a)    (_chk_io_ptr(a), *(volatile unsigned char _force *)(a))
#define _raw_readw(a)    (_chk_io_ptr(a), *(volatile unsigned short _force *)(a))
#define _raw_readl(a)    (_chk_io_ptr(a), *(volatile unsigned int _force *)(a))
```

2.5.5 内核空间到用户空间的映射

如果想在用户空间访问内核地址，可以采用 mmap 方法。应用程序通过内存映射可以直接访问设备的 I/O 存储区或 DMA 缓冲。映射一个设备，意味着使用用户空间的一段地址关联到设备内存上，这使得如果程序在分配的地址范围内进行读取或者写入，实际上就是对设备的访问。

```
unsigned long mmap (unsigned long addr, unsigned long len, int prot, int flags,
int fd, long offset);
内核到用户空间的映射接口
```

addr 是内存块的建议位置，不能确保 mmap()函数就一定使用这块内存区域，因此通常将其设

置成 NULL。len 是映射到调用进程地址空间的字节数，它从被映射文件开头 offset 个字节开始算起。prot 参数指定共享内存的访问权限。可取如下几个值的或操作：PROT_READ（可读），PROT_WRITE（可写），PROT_EXEC（可执行），PROT_NONE（不可访问）。flags 由以下几个常值指定：MAP_SHARED，MAP_PRIVATE，MAP_FIXED。其中，MAP_SHARED，MAP_PRIVATE 必选其一，而 MAP_FIXED 则不推荐使用。如果指定为 MAP_SHARED，则对映射的内存所做的修改同样影响到文件。如果是 MAP_PRIVATE，则对映射的内存所做的修改仅对该进程可见，对文件没有影响。Fd 是设备的文件描述符 offset 参数一般设为 0，表示从文件头开始映射。不是所有的设备可以进行 mmap 映射，比如串口和面向流的设备就无法进行。mmap 必须以 PAGE_SIZE 为单位进行映射，因此被映射的区域必须是 PAGE_SIZE 的整数倍。为了执行 mmap，驱动程序只需要为该地址范围建立合适的页表。可以使用 remap_page_range 函数一次性全部建立。相对于 Linux 2.4 来说，Linux 2.6 中的 remap_page_range 函数多了一个参数。虚拟内存区域（VMA）指针要作为第一个参数：

```
int remap_page_range(struct vm_area_struct *vma, unsigned long from, unsigned long
phys_addr, unsigned long size, pgprot_t prot)
```

Linux 2.6 下另一种实现内存映射的方法是使用 vm_operations，这种方法需要通过重载 VMA 操作来建立驱动特有的 nopage()方法。这种方法用于解决映射区域的页错误问题。nopage()方法不能用于映射 I/O 空间。只有系统内存区域能够使用这种映射方法。

```
struct vm operations struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    struct page * (*nopage)(struct vm_area_struct * area, unsigned long address,
    int *type);
    int (*populate)(struct vm_area_struct * area, unsigned long address, unsigned
    long len, pgprot_t prot, unsigned long pgoff, int nonblock);
#ifdef CONFIG_NUMA
    int (*set_policy)(struct vm_area_struct *vma, struct mempolicy *new);
    struct mempolicy *(*get_policy)(struct vm_area_struct *vma,unsigned long
    addr);
#endif
};
```

2.6　中断处理

2.6.1　硬件中断

在 Linux 下，硬件中断叫做 IRQ（Interrupt Request）。有两种 IRQ，即短类型和长类型。短 IRQ 需要很短的时间，在此期间机器的其他部分被锁定，而且没有其他中断被处理。一个长 IRQ 需要较长的时间，在此期间可能发生其他中断（但不是发自同一个设备）。

如果 CPU 接到一个中断，它就会停止一切工作（除非它正在处理一个更重要的中断，在这种情况下要等到更重要的中断处理结束后才会处理这个中断），把相关的参数存储到栈里，然后调用中断处理程序。这意味着在中断处理程序本身有些事情是不允许的，因为这时系统处在一个未知状态。解决这个问题的方法是让中断处理程序做需要马上做的事，通常是从硬件读取信息或给硬件发送信息立即返回，然后把对新信息的处理调度到以后去做。Linux 中的中断处理程序分为两个部分：上半部（tophalf）和下半部（bottom half）。之所以会有上半部和下半部之分，完全是考虑到中断处理的效率。

上半部的功能是响应中断。当发生中断时，它就把设备驱动程序中中断处理例程的下半部挂到该设备的下半部执行队列中去，然后继续等待新的中断到来。这样一来，上半部执行的速度就会很快，它就可以接受更多它负责的设备所产生的中断了。上半部之所以要快，是因为它是完全屏蔽中断的，如果它没有执行完，其他的中断就不能被及时地处理，只能等到这个中断处理程序执行完毕以后。所以，要尽可能多地对设备产生的中断进行服务和处理，中断处理程序就一定要快。

下半部的功能是处理比较复杂的过程。下半部和上半部最大的区别是下半部可中断，而上半部却不可中断。下半部几乎完成了中断处理程序所有的事情，因为上半部只是将下半部排到了它们所负责的设备的中断处理队列中去，然后就什么都不管了。下半部所负责的工作一般是查看设备以获得产生中断的事件信息，并根据这些信息（一般通过读设备上的寄存器得来）进行相应地处理。下半部是可中断的，所以在它运行期间，如果其他的设备产生了中断，这个下半部可以暂时地中断掉，等到那个设备的上半部运行完了，再回头来运行这个下半部。可以使用 tasklet 机制或软中断机制（softirq）来实现中断下半部处理，而 tasklet 则是基于 softirq 的。

Linux 内核中中断请求队列用 irq_desc 描述：

```
Struct irq_desc {
    Irq_flow_handler_t  handle_irq;
    Struct irq_chip     *chip;
    Void                *handler_data;
    Void                *chip_data;
    Struct irqaction    *action;    // IRQ 服务列表
    Unsigned int        status;     // IRQ 状态
    Unsigned int        depth;      // 中断嵌套禁止
    Unsigned int        wake_depth; // 嵌套唤醒允许
    Unsigned int        irq_count;  // 用来检测发生的中断数
    Unsigned int        irqs_unhandled;
    Spinlock_t      lock;
#ifdef CONFIG_SMP
    cpumask t       affinity;
    unsigned int    cpu;
#endif
#if defined(CONFIG GENERIC PENDING IRQ) || defined(CONFIG IRQBALANCE)
    cpumask t       pending mask;
#endif
#ifdef CONFIG_PROC_FS
```

```
    struct proc_dir_entry  *dir;
#endif
    const char  *name;
}
struct irq_desc irq_desc[NR_IRQS];
```


IRQ 服务列表的结构描述为：

```
Struct irqaction {
    Irq_handler_t handler;//设备中断处理函数
    Unsigned long flags;
    Cpumask t mask;
    Const char *name;
    Void *dev_id;//设备 ID
    Struct irqaction *next;
    Int irq;//中断号
    Struct proc_dir_entry *dir;
};
```

可以使用下面的函数申请一个中断：

```
int request_irq(unsigned int irq,void (*handler)(int irq,void dev_id,struct
pt_regs *regs), unsigned long flags, const char *device, void *dev_id);
```

参数 irq 表示所要申请的硬件中断号。handler 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 irq 为中断号，regs 为中断发生时寄存器的内容。device 为设备名，将会出现在/proc/interrupts 文件里。flags 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是中断处理程序是快速处理程序（flags 里设置了 SA_INTERRUPT）还是慢速处理程序（不设置 SA_INTERRUPT），快速处理程序运行时，所有中断都被屏蔽，而慢速处理程序运行时，除了正在处理的中断外，其他中断都没有被屏蔽。Dev_id 参数是设备 ID。这个参数通常设置为 NULL，但是，如果需要共享 IRQ，以使得稍后那个 IRQ 被 free_irq() 释放时，正确的设备会被放开，那么它需要是非空的。由于它是 void *，所以它可以指向任何内容，不过，通常的做法是传递驱动程序的设备结构体。

```
SA_INTERRUPT:快速中断处理。
SA_SHIRQ:表示中断可以在设备之间共享。
SA_SAMPLE_RANDOM:能对/dev/random 或/dev/urandom 设备使用的熵池有贡献。
```

使用下面的函数释放一个中断：

```
void free_irq(unsigned int irq, void *dev_id);
```

在 Linux 2.6 中，驱动程序如果要从一个设备上发出一个中断需要返回 IRQ_HANDLED，如果不是的话返回 IRQ_NONE。这样可以帮助内核的 IRQ 层清楚地识别出哪个驱动程序正在处理那个特定的中断。如果一个中断请求不断到来而且没有注册那个设备的处理程序（例如，所有的驱动程序都返回 IRQ_NONE），内核就会忽略来自那个设备的中断。默认情况下，驱动程序 IRQ

例程应该返回 IRQ_HANDLED，当驱动程序正在处理那个中断时却返回了 IRQ_NONE，说明存在错误。

禁止与允许中断的函数：

```
void disable irq(int irq);
void disable irq nosync(int irq);
//禁止单个中断
void enable irq(int irq);
//允许单个中断
void local_irq_save(unsigned long flags);
void local_irq_diable(void);
//禁止所有中断
void local_irq_restore(unsigned long flags);
void loval_irq_enable(void);
//允许所有中断
```

2.6.2 软中断机制

Linux 下的软中断机制在 linux/kernel/softirq.c 中实现。软中断是利用硬件中断的概念，用软件方式进行模拟，实现异步处理。软中断的行为用 softirq_action 描述。软中断存放在软中断矢量表中。

```
Struct softirq_action
{
    void    (*action)(struct softirq_action *);
    void    *data;
};
static struct softirq_action softirq_vec[32] _cacheline_aligned_in_smp;
                                                          //软中断矢量表
```

系统在 ksoftirqd 内核进程中调用_do_softirq 循环检测软中断是否产生：

```
Asmlinkage void  do softirq(void)
{
    struct softirq_action *h;
    _u32 pending;
    int max_restart = MAX_SOFTIRQ_RESTART;
    pending = local_softirq_pending();
    local_bh_disable();
restart:
    //允许中断之前先复位中断标志
    local softirq pending() = 0;
    local irq enable();
    h = softirq vec;
    do {
```

```
            if (pending & 1)
                h->action(h);//执行处理函数
            h++;
            pending >>= 1;
        } while (pending);
        local_irq_disable();
        pending = local_softirq_pending();
        if (pending && --max restart)
            goto restart;
        if (pending)
            wakeup softirqd();
         local bh enable();
    }
```

Linux 2.6 中定义了下列软中断优先级：

```
Enum
{
    HI_SOFTIRQ=0,// 下半部处理，高优先
    TIMER_SOFTIRQ,//定时器
    NET_TX_SOFTIRQ,//网络发送
    NET_RX_SOFTIRQ,//网络接收
    BLOCK_SOFTIRQ,//块层设备
    TASKLET_SOFTIRQ,//任务队列
    SCHED_SOFTIRQ,//调度
};
```

在软中断子系统初始化时启动下半部处理和任务队列：

```
void _init softirq_init(void)
{
    open softirq(TASKLET SOFTIRQ, tasklet action, NULL);
    open softirq(HI SOFTIRQ, tasklet hi action, NULL);
}
```

2.7 /proc 系统

Linux 内核提供了一种/proc 文件系统，可以在运行时访问内核内部数据结构、改变内核设置，用来向进程发送信息。这个虚拟文件系统可以和内核内部数据结构进行交互，获取有关进程的有用信息，在运行中改变设置（通过改变内核参数）。ps、top 命令就是通过读取/proc 下的文件来获取它们需要的信息。与其他文件系统不同，/proc 存在于内存之中，而不是硬盘上。/proc 由内核控制，没有承载/proc 的设备。因为/proc 主要存放由内核控制的状态信息，所以大部分这些信息的逻辑位置位于内核控制的内存。如果系统中还没有加载 proc 文件系统，可以通过如下命令加载 proc 文件系统：

```
mount -t proc proc /proc
```

下面介绍几个重要的/proc 文件，如表 2.4 所示：

表 2.4　/proc 目录下的文件

文件	说明
/proc/devices	这个文件列出所有字符设备和块设备的主设备号，以及对应这些设备号的设备名称
/proc/filesystems	这个文件列出可供使用的文件系统类型，一种类型占一行。它包括编入内核的文件系统和可加载的内核模块加入的其他文件系统类型
/proc/interrupts	这个文件的每一行都有一个保留的中断。每行中的域有：中断号，本行中断的发生次数，可能带有一个加号的域（SA_INTERRUPT 标志设置），以及登记这个中断的驱动程序的名字
/proc/ioports	这个文件列出了诸如磁盘驱动器，以太网卡和声卡设备等多种设备驱动程序登记的 I/O 端口范围
/proc/kmsg	这个文件用于检索用 printk 生成的内核消息。任何时刻只能有一个具有超级用户权限的进程可以读取这个文件，也可以用系统调用 syslog 检索这些消息。通常使用工具 dmesg 或守护进程 klogd 检索这些消息
/proc/ksyms	这个文件列出了已经登记的内核符号；这些符号给出了变量或函数的地址。每行给出一个符号的地址，符号名称以及登记这个符号的模块。程序 ksyms，insmod 和 kmod 使用这个文件。它还列出了正在运行的任务数，总任务数和最后分配的 PID
/proc/modules	这个文件给出可加载内核模块的信息。lsmod 程序用这些信息显示有关模块的名称、大小、使用数目方面的信息

另外通过/proc 文件系统不需要重新引导内核就可以更改运行中的内核的参数 。但是不能使用文本编辑工具来修改/proc 文件系统，因为内核时时刻刻在修改/proc 文件系统。每当更改 /proc 文件系统中的任何内容时，都应该使用 echo 命令，然后从命令行将输出重定向至 /proc 下所选定的文件中。例如：

```
#echo "Your-New-Kernel-Value" > /proc/your/file
```

类似的，如果希望查看 /proc 中的信息，应该使用专门用于此用途的命令，或者使用命令行下的 cat 命令。

```
#cat /proc/interrupts
```

使用/proc 的模块必须包含<linux/proc_fs.h>头文件。下面介绍两个重要函数：

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof, void *data);
//输出信息。page 为将要写入数据的缓冲区指针。start 为数据将要写入的页面位置。offset 为页面中
//的偏移量。count 为写入的字节数。eof 指向一个整型数，当没有更多数据时，必须设置这个参数，data
//为驱动程序特定的数据指针，可用于内部使用。函数的返回值表示实际放入页面缓冲区的数据字节数
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,struct
proc_dir_entry *parent);
创建目录。其中 name 为文件名称。mode 为文件权限。parent 为文件的父目录的指针，它为 NULL 时代
//表父目录为/proc
```

2.8 工作队列

Linux 2.6 中工作队列用于取代任务队列接口，它包括一系列将要执行的任务和执行这些任务的内核线程。每个工作队列有一个专门的线程，所有的任务必须在进程的上下文中运行，这样它们可以安全休眠。处于特定用途的工作队列之外的任务可以永远处于休眠状态，因为它们不需要与任何队列中的任务进行协作。驱动程序可以创建并使用它们自己的工作队列，或者使用内核的一个工作队列。

```
//创建工作队列，在这里 name 是工作队列的名字
struct workqueue_struct *create_workqueue(const char *name);
//在编译期初始化一个工作队列任务
DECLARE_WORK(name, void (*function)(void *), void *data);
//在运行期初始化一个工作队列
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

用下面的函数调用来把一个作业加入到工作队列中：

```
int fastcall queue_work(struct workqueue_struct *wq, struct work_struct *work);
int fastcall queue_delayed_work(struct workqueue_struct *wq,struct work_struct
*work, unsigned long delay);
```

在 queue_delayed_work()中指定 delay 是为了保证至少在经过一段给定的最小延迟时间以后，工作队列中的任务才可以真正执行。

工作队列中的任务由相关的工作线程执行，可能是在一个无法预期的时间内（取决于任务的处理时间、中断等因素），或者是在一段延迟以后。任何一个在工作队列中等待了无限长的时间也没有运行的任务可以用下面的方法取消：

```
int cancel_delayed_work(struct work_struct *work);
```

如果当一个取消操作的调用返回时，任务正在执行中，那么这个任务将继续执行下去，但不会再加入到队列中。

```
void fastcall flush_workqueue(struct workqueue_struct *wq);
//清空工作队列中的所有任务
void destroy_workqueue(struct workqueue_struct *wq);
//销毁工作队列
int fastcall schedule_work(struct work_struct *work);
//向工作队列中添加一个任务
int fastcall schedule_delayed_work(struct work_struct *work, unsigned long delay);
//向工作队列中添加一个任务并延迟执行
```

当模块被卸载时应该去调用一个 flash_scheduled_work() 函数，这个函数会使等待队列中所有的任务都被执行。

2.9 异步 I/O

Linux 上的 AIO 在 Linux 2.5 版本的内核中首次出现，现在已经是 Linux 2.6 版本内核的一个标准特性了。异步 I/O（<linux/aio.h>）允许用户进程发起多个 I/O 操作而不用等待它们中的任何一个完成，操作的状态可以在稍后的时间获取。块设备与网络设备驱动的操作全是异步的。但是对字符型设备，需要在驱动程序中实现相应的异步函数，才能支持异步操作。在新的 file_operation 结构中，可以看到这方面的变化：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char _user *, size_t, loff_t);
    ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char _user *, size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t(*readv)(struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t(*writev)(struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
    unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
};
```

支持 AIO 的驱动要实现 aio_read、aio_write 等函数。内核中关于 AIO 的结构如下：

```
struct kiocb {
    struct list_head ki_run_list;
    long ki_flags;
    int ki_users;
```

```
    unsigned  ki key;
    struct file *ki_filp;
    struct kioctx   *ki_ctx;
    int (*ki_cancel)(struct kiocb *, struct io_event *);
    long (*ki_retry)(struct kiocb *);
    void (*ki_dtor)(struct kiocb *);
    struct list_head ki_list;//用于取消 AIO 的核心结构
    union {
        void _user *user;
        struct task_struct  *tsk;
    } ki obj;
    _u64    ki_user_data;//用户数据
    loff_t  ki_pos;
    void    *private;
};
```

在某些时候，同步特性是必需的。同步 iocb 允许 AIO 子系统在必要的时候被同步地使用。可以用下列函数判断请求是否需要做同步处理。当 AIO 为同步，返回真。

```
is_sync_kiocb(struct kiocb *iocb);
//是否该操作必须采用同步操作完成
ssize_t fastcall wait_on_sync_kiocb(struct kiocb *iocb);
//等待一个同步 iocb 完成
```

如果 AIO 操作完成，可以使用下列函数通知 AIO 子系统：

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

取消 AIO 操作需要自定义一个 ki_cancel 函数：

```
int my_aio_cancel(struct kiocb *iocb, struct io_event *event);
iocb->ki_cancel = my_aio_cancel;
```

在应用层，传输操作通过 AIOCB（AIO I/O Control Block）结构完成。AIO 接口函数非常简单，但是它为数据传输提供了必需的功能，并给出了两个不同的通知模型。下面是 AIO 的应用层接口函数：

```
int aio_read (struct aiocb *_aiocbp);
//请求异步读操作
int aio_write (struct aiocb *_aiocbp);
//请求异步写操作
int lio_listio (int _mode,struct aiocb *_const _list[_restrict_arr],int _nent,
struct sigevent *_restrict _sig);
//发起一系列 I/O 操作
int aio_error (_const struct aiocb *_aiocbp);
//检查异步请求的状态
_ssize_t aio_return (struct aiocb *_aiocbp);
```

```
//获得完成的异步请求的返回状态
int aio_cancel (int _fildes, struct aiocb *_aiocbp);
//取消异步 I/O 请求
int aio_suspend (_const struct aiocb *_const _list[], int _nent,_const struct
timespec *_restrict _timeout);
//挂起调用进程，直到一个或多个异步请求已经完成（或失败）
```

下面的代码演示了如何在程序中使用 AIO：

```
int aio_fd; //一个支持 AIO 的设备描述符
struct aiocb my aiocb;
struct sigaction sig act;
 //填充 struct aiocb
 my_aiocb.aio_fildes = aio_fd;
 my_aiocb.aio_nbytes = 256;
 my_aiocb.aio_offset = 0;
 my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
 my_aiocb.aio_sigevent.sigev_signo = SIGIO;
 my aiocb.aio sigevent.sigev value.sival ptr = &my aiocb;
 sigemptyset(&sig act.sa mask);
 sig act.sa flags = SA SIGINFO;
 //设置信号处理函数
 sig act.sa sigaction = sig handler;
 const struct aiocb *cblist[3] ={&my_aiocb,NULL,NULL};
 sigaction( SIGIO, &sig_act, NULL );
 //发起一个读操作
ret = aio_read( &my_aiocb );
 //等待信号出现
 while(aio_suspend(cblist, 3, NULL ) == 0)
 {
       ret = aio read( &my aiocb );
 };
```

接收处理过程如下：

```
void sig_handler(int signo, siginfo_t *info, void *context )
{
     int ret;
     struct aiocb *req;
     char *p=NULL;
     if (info->si_signo == SIGIO)
     {
               req = (struct aiocb *)info->si_value.sival_ptr;
               if (aio_error( req ) == 0) //检查错误
                 {
                     ret = aio_return( req );
```

```
                    p=(char*)req->aio_buf+ret;
                    *p=0;
                   printf("receive:%s\n",req->aio_buf);
               }
           }
           return;
   }
```

2.10 DMA

直接内存存取（DMA）是解决快速数据访问的有效方法。DMA 控制器可以在不受处理器干预的情况下在设备和系统内存之间高速传输数据。为了初始化数据传输，设备驱动将设置 DMA 通道地址和记数寄存器以描述数据传输方向以及读写类型。然后通知设备可以在任何时候启动 DMA 操作。传输结束时设备将产生中断。在传输过程中 CPU 可以转去执行其他任务。

Linux 中的 DMA 通道在各种处理器下有不同的定义，下面是 arch/arm/kernel/dma.c 文件中的定义：

```
struct dma_struct {
    struct scatterlist buf;      //单个 DMA
    int      sgcount;            //DMA  SG 的数量
    struct scatterlist *sg;      //分散收集列表，解决多页的 DMA 传输问题
    unsigned int    active:1;    //传输激活状态
    unsigned int    invalid:1;   //传输地址活字节数发生变化
    unsigned int    using_sg:1;  // 是否使用 scatterlist
    dmamode_t    dma_mode;       //DMA 模式
    int      speed;              //DMA 速度
    unsigned int    lock;        //设备已分配
    const char  *device_id;      //设备名
    unsigned int    dma_base;    //控制器基地址
    int      dma_irq;            //控制器中断号
    struct scatterlist cur_sg;   //当前控制器缓冲
    unsigned int    state;       //当前状态
    struct dma_ops  *d_ops;      //DMA 操作接口
};
static dma_t dma_chan[MAX_DMA_CHANNELS];
```

DMA 通道的设置通过下面的函数完成：

```
int request_dma(dmach_t channel, const char *device_id);
//获取 DMA 通道的使用权
void free_dma(dmach_t channel);
//释放 DMA 通道的使用权
void set_dma_count (dmach_t channel, unsigned long count);
```

```
//设置传输的字节数
void set_dma_addr (dmach_t channel, unsigned long physaddr);
//设置 DMA 传输的总线地址
void set_dma_speed(dmach_t channel, int cycle_ns);
//设置传输速度
void set_dma_sg (dmach_t channel, struct scatterlist *sg, int nr_sg);
//设置分散收集列表
void enable_dma (dmach_t channel);
//允许某个通道的 DMA 传输
void disable_dma (dmach_t channel);
//禁止某个通道的 DMA 传输
```

内核提供一组 DMA 映射函数，用于分配一个 DMA 缓冲区，并为该缓冲生成一个能够被设备访问的地址的组合。Linux 的 DMA 映射函数分为连续映射与流式映射两种。连续映射方式保证对处理器与 DMA 器件是一致的，而不会包含高速缓冲带来的问题。它最常用于持续的双向 I/O 缓冲。流式映射可能会包含高速缓冲带来的问题，常用于单一的传输过程。很少有驱动使用流式映射，流式映射主要用来兼容那些不能产生连续内存映射的老平台。

连续 DMA 内存分配与释放操作函数如下：

```
void *dma_alloc_coherent(struct device *dev, size_t size,dma_addr_t *dma_handle,
int flag);
void dma_free_coherent(struct device *dev, size_t size,void *cpu_addr, dma_addr_t
dma_handle);
```

流式 DMA 内存分配与释放操作函数如下:

```
void *dma_alloc_noncoherent(struct device *dev, size_t size,dma_addr_t *dma_handle,
int flag);
void dma_free_noncoherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t
dma_handle)
```

2.11 platform 概念

Linux 2.6 中引入了新的设备管理机制，所有的设备都被纳入 kobject 对象统一管理。系统中的任一设备在设备模型中都由一个 device 对象（struct device）描述，系统中的每个驱动程序由一个 device_driver 对象描述。每个 device 对象对应一个 device_driver 对象。它们对应的操作函数有：

```
int device_register(struct device * dev);//注册一个设备对象
void device_unregister(struct device * dev);//注销一个设备对象
int driver_register(struct device_driver * drv); //注册一个驱动对象
void driver_unregister(struct device_driver * drv); //注销一个驱动对象
```

平台设备概念的引入是能够更好地描述设备的资源信息。平台设备是系统中自治的实体，包括基于端口的设备、外围总线和集成入片上系统平台的大多数控制器，它们通常直接通过 CPU 的

总线寻址。每个平台设备被赋予一个名称，并分配一定数量的资源。平台的设备使用 struct platform_device 描述：

```
struct platform device {
    const char  * name;//名称
    u32     id;
    struct device   dev;//对应的设备
    u32     num_resources;//资源的数量
    struct resource * resource;//资源信息
};
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*suspend_late)(struct platform_device *, pm_message_t state);
    int (*resume_early)(struct platform_device *);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
};
```

name 是设备的识别标志。当使用 driver_register 注册一个驱动时，系统会到注册的设备表中寻找命名为 name 的设备，如果没有发现这个设备，注册将会失败。平台驱动遵循标准的驱动传统，当枚举在驱动之外处理，驱动必须提供 probe 和 remove 方法。必须在 probe 中确保设备资源的可用性。另外，平台驱动还支持电源管理和设备停止事件。与上面的函数相对应，内核提供了多平台设备的如下操作函数接口：

```
int platform_device_register(struct platform_device *);//注册一个平台设备
void platform_device_unregister(struct platform_device *);//注销一个平台设备
int platform_add_devices(struct platform_device **pdevs, int ndev);
                                                        //批量添加平台设备
int platform_driver_register(struct platform_driver *drv);
void platform_driver_unregister(struct platform_driver *drv);
```

而从内核代码中可以看出平台驱动和 device_driver 对象其实是很类似的。platform_driver_register 和 platform_driver_unregister 本质上就是 driver_register 和 driver_unregister。对于驱动开发人员，最重要的是 struct platform_device 中的 struct resource：

```
struct resource {
    const char *name;//名称
    unsigned long start, end;//起始与终止地址
    unsigned long flags;//标志
    struct resource *parent, *sibling, *child;
};
```

在 struct platform_device 中你可以设置多种资源信息。资源的 flags 标志包括：

```
#define IORESOURCE_IO   0x00000100  //I/O 资源
#define IORESOURCE_MEM  0x00000200  //内存资源
#define IORESOURCE_IRQ  0x00000400  //中断资源
#define IORESOURCE_DMA  0x00000800  //DMA 资源
```

内核提供了一组函数，用来获取设备的资源信息：

```
struct resource * platform_get_resource(struct platform_device *dev, unsigned int
type, unsigned int num);
//根据资源类型和序号来获取指定的资源
struct int platform_get_irq(struct platform_device *dev, unsigned int num);
//根据序号获取资源的中断号
struct resource * platform_get_resource_byname(struct platform_device *dev,
unsigned int type, char *name);
//根据名称和类别获取指定的资源
int platform_get_irq_byname(struct platform_device *dev, char *name);
//根据名称获取资源中的中断号
```

以后会发现以上函数对编写驱动程序很有用。

2.12 简单驱动例程

2.12.1 信号量同步

下面为使用信号量实现一个同步读写的例子。基本的程序见第 1 章，只改动了 read、write 相关的实现部分。首先将信号量加入到自定义的设备结构中：

```
struct DEMO dev
{
    struct semaphore sem;
    struct cdev cdev;
};
struct DEMO_dev *DEMO_devices;
```

在模块初始化时对信号量进行初始化：

```
int DEMO init module(void)
{
    init MUTEX(&DEMO devices->sem);
   ...
}
```

现在就可以调用信号量保护临界资源，而不用担心多进程访问造成的冲突：

```
ssize_t DEMO_read(struct file *filp, char _user *buf, size_t count,loff_t *f_pos)
```

```
{
    struct DEMO_dev *dev = filp->private_data;
    int result;
    loff_t pos;
   //获取信号量
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    ...
  out:
   //释放信号量，获取信号量和释放信号量必须成对出现
    up(&dev->sem);
    return count;
}
```

2.12.2 阻塞式读写

这个例子采用等待队列来实现阻塞式访问。它的基本步骤与信号量非常相似。首先将等待队列加入到自定义的设备结构中：

```
struct DEMO_dev
{
    struct semaphore sem;
    wait_queue_head_t wq;
    struct cdev cdev;
};
struct DEMO_dev *DEMO_devices;
```

下面需要对等待队列进行初始化：

```
int DEMO_init_module(void)
{
   ...
    init_waitqueue_head(&DEMO_devices->wq);
   ...
}
```

现在就可以调用等待队列来实现阻塞式读写：

```
ssize_t DEMO_read(struct file *filp, char __user *buf, size_t count,loff_t *f_pos)
{
    struct DEMO_dev *dev = filp->private_data;
    //等待数据可获得
    if(wait_event_interruptible(dev->wq, flag != 0))
    {
        return - ERESTARTSYS;
```

```
    }
    flag = 0;
    //信号量不能放到wait_event_interruptible之前。因为获得信号量后，如果没有数据可读，
    //则会造成程序死锁
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
   //将数据复制到用户内存
  out:
    up(&dev->sem);
    return count;
}
```

需要在驱动中其他地方，如中断、定时器等处理函数中唤醒等待队列。这里通过写操作唤醒等待队列。

```
ssize_t DEMO_write(struct file *filp, const char _user *buf, size_t count,loff_t
*f_pos)
{
    struct DEMO_dev *dev = filp->private_data;
    ssize t retval = -ENOMEM;
   //可中断的获取锁
    if (down_interruptible(&dev->sem))
    {
        return -ERESTARTSYS;
    }
      ...
    up(&dev->sem);
    flag = 1;
    //通知数据可获得
    wake_up_interruptible(&dev->wq);
    return count;
  out:
    up(&dev->sem);
    return retval;
}
```

下面编写两个测试程序，一个负责读，一个负责写，代码见光盘。测试结果如下：

~$insmod demo.ko

Using demo.ko

~$mknod /dev/fgj c 224 0

~$./read &

[1] 335

~$./write

The data is FG

2.12.3 定时器

这个例子使用定时器进行 10 次简单输出，超过 10 次后删除定时器。

```
static struct timer_list simple_timer;
#define SIMPLE_TIMER_DELAY 2*HZ //2 秒
```

对定时器进行初始化：

```
int DEMO init module(void)
{
        ...
    init timer(&simple timer);
    simple_timer.function = &simple_timer_handler;//处理函数
    simple_timer.expires = jiffies + SIMPLE_TIMER_DELAY;//到期时间
    add_timer (&simple_timer);
    return 0;
fail:
    DEMO_cleanup_module();
    return result;
}
```

下面是定时器的处理函数。注意定时器属于软中断，不要使用不可打断的信号量获取函数来获取信号量。

```
static void simple timer handler( unsigned long data)
{
    simple_timer.expires    = jiffies + SIMPLE_TIMER_DELAY;
    simple timer.function   = &simple timer handler;
    add_timer (&simple_timer);
    if(Condition<10)
    {
        printk("the %d times\n",Condition);
        Condition++;
    }
    else
    {
        printk("delete timer\n");
        //删除活动的定时器，如果定时器到期，会自动被删除
        del_timer(&simple_timer);
    }
    return ;
}
```

2.12.4 内存映射

这个例子改自 Martin Frey 的 mmap 程序。它实现了 kmalloc 和 vmalloc 分配内存的映射方法。最终的结果是应用程序通过 mmap 方法，正确地访问了驱动中分配的内存区域。首先定义四个指针：

```
static int *vmalloc_area = NULL;    //页对齐的区域
static int *vmalloc_ptr = NULL;     //未对齐的区域
static int *kmalloc_area = NULL;    //页对齐的区域
static int *kmalloc_ptr = NULL;     //未对齐的区域
```

在驱动初始化的时候进行内存分配，并将分配的内存区域标记为保留：

```
int DEMO init module(void)
{
    ...//字符驱动注册,见第 1 章
    kmalloc_ptr=kmalloc(LEN+2*PAGE_SIZE, GFP_KERNEL);
    //页对齐处理
    kmalloc_area=(int *)(((unsigned long)kmalloc_ptr + PAGE_SIZE -1) & PAGE_MASK);
    //标记为保留，保留页被锁定，可以安全地映射到用户空间
    for(virt addr=(unsigned long)kmalloc area;
                virt addr<(unsigned long)
                kmalloc area+LEN;
                virt addr+=PAGE SIZE;
{
        SetPageReserved(virt_to_page(virt_addr));
}
vmalloc_ptr=vmalloc(LEN+2*PAGE_SIZE);
vmalloc_area=(int *)(((unsigned long)vmalloc_ptr + PAGE_SIZE -1) & PAGE_MASK);
for (virt_addr=(unsigned long)vmalloc_area;virt_addr<
    (unsigned long)(&(vmalloc_area[LEN/sizeof(int)]));virt_addr+=PAGE_SIZE)
{
    SetPageReserved(virt_to_page(virt_to_kseg((void *)virt_addr)));//标记为保留
}
//对内存区域赋值
for (i=0; i<(LEN/sizeof(int)); i+=2)
{
     vmalloc_area[i]=(0xaffe<<16) + i;
     vmalloc_area[i+1]=(0xbeef<<16) + i;
     kmalloc_area[i]=(0xdead<<16) +i;
     kmalloc_area[i+1]=(0xbeef<<16) + i;
  }
printk("vmalloc_area at 0x%p (phys 0x%lx)\n", vmalloc_area,
             virt_to_phys((void *)virt_to_kseg(vmalloc_area)));
```

```
printk("kmalloc area at 0x%p (phys 0x%lx)\n", kmalloc_area,
            virt_to_phys((void *)virt_to_kseg(kmalloc_area)));
}
```

在驱动卸载的时候释放内存：

```
void DEMO_cleanup_module(void)
{
      ...
       unsigned long virt_addr;
       for(virt_addr=(unsigned long)kmalloc_area;
            virt_addr<(unsigned long)kmalloc_area+LEN; virt_addr+=PAGE_SIZE)
       {
            ClearPageReserved (virt_to_page(virt_addr));//取消页保留
       }
       for (virt_addr=(unsigned long)vmalloc_area;
         virt_addr<(unsigned long)(&(vmalloc_area[LEN/sizeof(int)]));
            virt_addr+=PAGE_SIZE)
       {
            ClearPageReserved (virt_to_page(virt_to_kseg((void *)virt_addr)));
       }
       if (vmalloc_ptr)vfree(vmalloc_ptr);
       if (kmalloc_ptr)kfree(kmalloc_ptr);
}
```

再在你的驱动中加入 mmap 接口：

```
struct file_operations DEMO_fops = {
    .owner =    THIS_MODULE,
    .open =     DEMO_open,
    .mmap =     DEMO_mmap,
    .release = DEMO_release,
};
//映射内存就是创建所需要的页表
static int DEMO_mmap(struct file *filp, struct vm_area_struct *vma)
{
       unsigned long offset = vma->vm_pgoff<<PAGE_SHIFT;
       unsigned long size = vma->vm_end - vma->vm_start;
       //判断偏移是否页对齐
       if (offset & ~PAGE_MASK)
       {
              printk("offset not aligned: %ld\n", offset);
              return -ENXIO;
       }
       //判断尺寸是否过大
       if (size>LEN)
```

```
        {
                printk("size too big\n");
                return(-ENXIO);
        }
        //仅支持 VM_SHARED
    if ((vma->vm_flags & VM_WRITE) && !(vma->vm_flags & VM_SHARED))
    {
         printk("writeable mappings must be shared, rejecting\n");
         return(-EINVAL);
    }
     //不希望区域被置换出去，将其锁定
    vma->vm flags |= VM LOCKED;
  //对于 vmalloc 分配的连续虚拟地址区域，采用页错误处理方法。当应用程序第一次访问该页时，
  //页错误处理方法会计算正确的物理页的位置
  //对于物理上连续的页，使用 remap_page_range 在物理页和虚拟地址之间创建一个映射
        if (offset == 0)
        {
                vma->vm_ops = &mmap_drv_vm_ops;
                mmap drv vopen(vma);
        } else if (offset == LEN)
        {
                if (remap_page_range(vma,vma->vm_start,virt_to_phys((void*)
                ((unsigned long)kmalloc area)),size,PAGE SHARED))
                {
                        printk("remap page range failed\n");
                        return -ENXIO;
                }
        } else
        {
                printk("offset out of range\n");
                return -ENXIO;
        }
        return(0);
}
```

下面是测试程序代码：

```
int main(void)
{
  int fd;
  unsigned int *vadr;
  unsigned int *kadr;
  if ((fd=open("/dev/fgj", O RDWR))<0)
    {
     perror("open failed");
     exit(-1);
```

```
    }
   //映射操作函数
  vadr = mmap(0, LEN, PROT_READ, MAP_SHARED, fd, 0);
  if (vadr == MAP_FAILED)
  {
        perror("mmap");
        exit(-1);
  }
  else
  {
      printf("mmap ok\n");
  }
  printf("vadr0 0x%x vadr1 0x%x\n", vadr[0], vadr[1]);
  printf("vadr[LEN/sizeof(int)-2] 0x%x vadr[LEN/sizeof(int)-1] 0x%x\n",
                      vadr[LEN/sizeof(int)-2], vadr[LEN/sizeof(int)-1]);
//与上面的区别在于偏移量(最后一个参数)
  kadr = mmap(0, LEN, PROT_READ, MAP_SHARED, fd, LEN);
  if (kadr == MAP_FAILED)
  {
        perror("mmap");
        exit(-1);
  }
  else
  {
          printf("mmap ok\n");
  }
  printf("kadr0 0x%x kadr1 0x%x\n", kadr[0], kadr[1]);
  printf("kadr[LEN/sizeof(int)-2] 0x%x kadr[LEN/sizeof(int)-1] 0x%x\n",
                       kadr[LEN/sizeof(int)-2], kadr[LEN/sizeof(int)-1]);
  close(fd);
  return(0);
}
```

测试结果如下：

[root@(none) tmp]# insmod demo.ko

Using demo.ko

vmalloc_area at 0xC4959000 (phys 0x3256D000)

kmalloc_area at 0xC25A0000 (phys 0x416B6000)

[root@(none) tmp]# mknod /dev/fgj c 224 0

[root@(none) tmp]# ./read

mmap ok

mmap_drv: page fault for offset 0x0 (kseg xc256D000)

mmap_drv: page fault for offset 0xF000 (kseg xc258A000)

vadr0 0XAFFE0000 vadr1 0xBEEF0000

vadr[LEN/sizeof(int)–2] 0xAFFE3FFE vadr[LEN/sizeof(int)–1] 0xBEEF3FFE
mmap ok
kadr0 0xDEAD0000 kadr1 0xBEEF0000
kadr[LEN/sizeof(int)–2] 0xDEAD3FFE kadr[LEN/sizeof(int)–1] 0xBEEF3FFE
[root@(none) tmp]#

2.12.5 /proc 访问

这个例子演示了如何访问/proc 文件系统的文件。

```
int init_demo_module( void )
{
  int ret = 0;
  DEMO_buffer = (char *)vmalloc( MAX_DEMO_LENGTH );
  if (!DEMO_buffer)
 {
    ret = -ENOMEM;
 }
else
 {
    memset( DEMO_buffer, 0, MAX_DEMO_LENGTH );
     //在/proc 文件系统中创建节点
    proc_entry = create_proc_entry( "demo", 0644, NULL );
    if (proc_entry == NULL)
     {
      ret = -ENOMEM;
      vfree(DEMO_buffer);
      printk(KERN_INFO "demo: Couldn't create proc entry\n");
    }
    else
    {
      DEMO_index = 0;
      next_demo = 0;
      //设置访问接口
      proc_entry->read_proc = demo_read;
      proc_entry->write_proc = demo_write;
      proc_entry->owner = THIS_MODULE;
      printk(KERN_INFO "demo: Module loaded.\n");
    }
  }
  return ret;
}
void cleanup_demo_module( void )
{
```

```
    //清除节点
  remove_proc_entry("demo", &proc_root);
  vfree(DEMO_buffer);
  printk(KERN_INFO "demo: Module unloaded.\n");
}
```


读接口的实现：

```
int demo_read( char *page, char **start, off_t off,int count, int *eof, void *data )
{
  int len;
  if (off > 0) {
    *eof = 1;
    return 0;
  }
  if (next_demo >= DEMO_index) next_demo = 0;
  len = sprintf(page, "%s\n", &DEMO_buffer[next_demo]);
  next_demo += len;
  return len;
}
```

写接口的实现：

```
ssize_t demo_write( struct file *filp, const char _user *buff,unsigned long len,
void *data )
{
  int space_available = (MAX_DEMO_LENGTH-DEMO_index)+1;
  if (len > space_available) {
    printk(KERN_INFO "demo: DEMO buffer is full!\n");
    return -ENOSPC;
  }
  if (copy from user( &DEMO buffer[DEMO index], buff, len )) {
    return -EFAULT;
  }
  DEMO index += len;
  DEMO buffer[DEMO index-1] = 0;
  return len;
}
```

测试结果如下：

[root@(none) tmp]# insmod demo.ko

Using demo.ko

demo: Module loaded.

[root@(none) tmp]# cd /proc

[root@(none) proc]# ls

```
1          292          cpu            ide           modules      sys
12         293          cpuinfo        interrupts    mounts       sysvipc
13         298          demo           iomem         mtd          tty
2          3            devices        ioports       net          uptime
24         322          diskstats      kallsyms      partitions   ersion
25         332          driver         kmsg          scsi         vmstat
258        4            execdomains    loadavg       self
26         buddyinfo    fb             locks         slabinfo
269        bus          filesystems    meminfo       stat
27         cmdline      fs             misc          swaps
[root@(none) proc]# echo "first" >/proc/demo
[root@(none) proc]# echo "second" >/proc/demo
[root@(none) proc]# cat /proc/demo
first
[root@(none) proc]# cat /proc/demo
second
[root@(none) proc]#
```

2.12.6 工作队列

这个例子是关于工作队列的。它是在 2.12.2 小节的基础上修改而成的，它演示了如何使用工作队列启动一个任务。首先定义一个工作队列：

```
static struct work struct task;
static struct workqueue_struct *my_workqueue;
```

对工作队列进行初始化：

```
my_workqueue = create_workqueue("MYQUENU");
//设置工作队列的处理函数，并向工作队列传递参数 DEMO_devices
INIT WORK(&task,DemoTask, DEMO devices);
queue_work(my_workqueue, &task);
```

在处理函数中修改内存区域，唤醒等待队列。

```
static void DemoTask(void *p)
{
    struct DEMO_dev *dev =(struct DEMO_dev *)p;
     flag = 1;
     memset(demoBuffer,0x31,256);
     wake_up_interruptible(&dev->wq); //唤醒读进程
}
```

测试程序很简单：

```
main()
{
    int fd;
    char num[2];
    fd = open("/dev/fgj", O RDWR, S IRUSR | S IWUSR);
    if (fd != - 1)
    {
        read(fd, &num,2);
        num[2]=0;
        printf("The data is %s\n", num);
    }
    else
    {
        printf("device open failure\n");
    }
    close(fd);
}
```

测试结果如下：

[root@(none) tmp]# insmod demo.ko

Using demo.ko

[root@(none) tmp]# mknod /dev/fgj c 224 0

[root@(none) tmp]# rz

..[root@(none) tmp]# .**B0100000023BE50

[root@(none) tmp]# ./read

The data is 11

第3章 GPIO 驱动

ARM 公司自 1990 年正式成立以来，在 32 位 RISC（Reduced Instruction Set Computer）CPU 开发领域不断取得突破，其结构已经从 V3 发展到 V6。目前非常流行的 ARM 芯核有 ARM7TDMI，ARM720T，ARM9TDMI，ARM920T，ARM940T，ARM926EJ-S，ARM1020E 和 XScale 等。从本章开始，将开始走进 ARM 驱动程序开发的世界，领略其独特的魅力。

3.1 ARM 体系结构概述

3.1.1 RISC 结构

传统的 CISC（Complex Instruction Set Computer，复杂指令集计算机）结构有其固有的缺点，即指令结构复杂，指令使用频率却相差悬殊。基于以上的不合理性，1979 年美国加州大学伯克利分校提出了 RISC（Reduced Instruction Set Computer，精简指令集计算机）的概念，RISC 并非只是简单地去减少指令，而是把重点放在了如何使计算机的结构更加简单合理地提高运算速度上。RISC 结构优先选取使用频率最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻址方式种类减少；以控制逻辑为主，不用或少用微码控制等措施来达到上述目的。ARM 体系结构就是建立在 RISC 的基础上，它具有 RISC 的基本特点：

（1）采用固定长度的指令格式。

（2）寻址方式简单。

（3）使用单周期指令，便于流水线操作执行。

（4）大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，以提高指令的执行效率。

此外 ARM 体系还引进了一些新的技术：

（1）所有的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率。

（2）可用加载/存储指令批量传输数据，以提高数据的传输效率。

（3）可在一条数据处理指令中同时完成逻辑处理和移位处理。

（4）在循环处理中使用地址的自动增减来提高运行效率。

从编程的角度看，ARM 微处理器的工作状态一般有两种，并可在这两种状态之间切换：第一种为 ARM 状态，此时处理器执行 32 位的字对齐的 ARM 指令；第二种为 Thumb 状态，此时处理器执行 16 位的、半字对齐的 Thumb 指令。ARM 微处理器一般都支持两种指令集：ARM 指令

集和 Thumb 指令集。其中，Thumb 指令集为 ARM 指令集的功能子集，比 ARM 指令集更节约存储空间。

ARM 微处理器的存储器格式包括两种格式：（1）大端格式：字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中；（2）小端格式：与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。可以采用如下方法，判断处理器的存储器格式：

```
int GetEndianness()
{
   short s = 0x0110;
   char *p = (char *) &s;
   if (p[0] == 0x10)
      return 0;// 小端格式
   else
      return 1;// 大端格式
}
```

3.1.2 处理器模式

ARM 微处理器支持 7 种运行模式，分别如下所示。

（1）用户模式（User）：ARM 处理器正常的程序执行状态。

（2）快速中断模式（FIQ）：用于高速数据传输或通道处理。

（3）外部中断模式（IRQ）：用于通用的中断处理。

（4）管理模式（Supervisor）：操作系统使用的保护模式。

（5）指令终止模式（Abort）：当指令预取终止时进入该模式。

（6）系统模式（System）：运行具有特权级别的操作系统任务。

（7）未定义模式（Undefined）：系统遇到一个未定义的指令。

运行模式的改变可以通过软件设置，或由外部中断以及异常导致。大部分的应用程序在用户模式执行，非用户模式用于处理中断和异常，以及访问受保护的资源。

3.1.3 寄存器组织

ARM 微处理器在 ARM 状态和 THUMB 状态下的寄存器组织是不一样的。ARM 状态下有 31 个通用寄存器和 6 个状态寄存器。31 个通用寄存器包括 R0～R15，可以分为三类：（1）未分组（Unbanked）寄存器 R0～R7；（2）分组（Banked）寄存器 R8～R14；（3）程序计数器 PC（R15）。未分组寄存器在各种模式下是相同的，分组寄存器在各种模式下是不同的。状态寄存器包括 CPSR（Current Program Status Register，当前程序状态寄存器）和 SPSR（Saved Program Status Register，备份的程序状态寄存器）。CPSR 可在任何运行模式下被访问。当异常发生时，SPSR 用于保存 CPSR 的当前值，从异常退出时可由 SPSR 来恢复 CPSR。图 3.1 和图 3.2 是 ARM 模式下寄存器。其中

带三角阴影的是分组寄存器。

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

图 3.1　ARM 模式下的通用寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

图 3.2　ARM 模式下的程序状态寄存器

程序状态寄存器用来保存 ALU 中的当前操作信息、控制允许和禁止中断、设置处理器的运行模式。图 3.3 给出了 CPSR 各位的作用。表 3.1 给出了模式位 M[4：0]的具体含义。

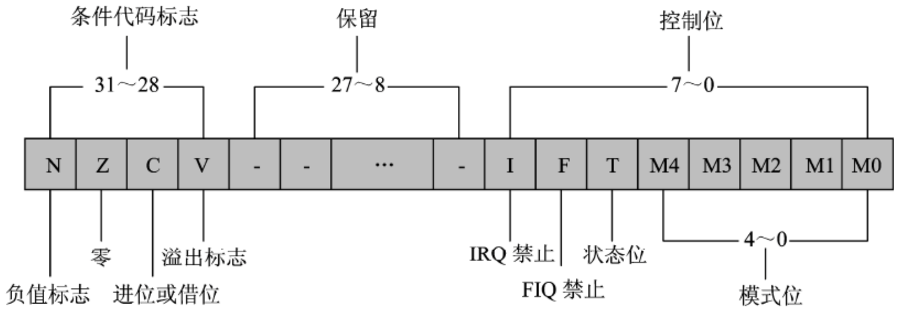


图 3.3　程序状态寄存器 CPSR

表 3.1　模式位 M[4：0]的具体含义

M[4：0]	处理器模式	可访问的寄存器
10000	用户模式	PC，CPSR，R0～R14
10001	FIQ 模式	PC，CPSR，SPSR_fiq，R14_fiq-R8_fiq，R7～R0
10010	IRQ 模式	PC，CPSR，SPSR_irq，R14_irq，R13_irq，R12～R0

续表

M[4：0]	处理器模式	可访问的寄存器
10011	管理模式	PC，CPSR，SPSR_svc，R14_svc，R13_svc，R12～R0
10111	中止模式	PC，CPSR，SPSR_abt，R14_abt，R13_abt，R12～R0
11011	未定义模式	PC，CPSR，SPSR_und，R14_und，R13_und，R12～R0
11111	系统模式	PC，CPSR（ARM V4 及以上版本），R14～R0

3.1.4 异常处理

ARM 体系中的异常处理包括中断请求处理、程序中止以及未定义等。在处理异常之前，当前处理器的状态必须保留，这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按照固定的优先级进行处理，复位异常的优先级别最高。ARM 体系结构所支持的异常如表 3.2 所示：

表 3.2 ARM 体系结构所支持的异常

异常类型	名称	优先级	具体含义
Reset	复位	1（最高）	当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处理程序处执行
Undefined	未定义指令	6	当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常。可使用该异常机制进行软件仿真
SWI	软件中断	6	该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令。可使用该异常机制实现系统功能调用
Prefech Abort	指令预取中止	5	若处理器预取指令的地址不存在，或该地址不允许当前指令访问，存储器会向处理器发出中止信号，但当预取的指令被执行时，才会产生指令预取中止异常
Data Abort	数据中止	2	若处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常
IRQ	外部中断请求	4	当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务
FIQ	快速中断请求	3	当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常

异常处理程序一般是放在异常向量表中。异常向量表实际上是一张地址跳转表。异常发生时，系统从对应的异常向量表查询目标地址，然后跳转到该地址执行。

```
b   ResetHandler ;handle for reset              //0x00000000
b   HandlerUndef ;handler for Undefined mode    //0x00000004
b   HandlerSWI ;handler for SWI interrupt       //0x00000008
b   HandlerPabort ;handler for PAbort           //0x0000000C
b   HandlerDabort ;handler for DAbort           //0x00000010
b   .            ;reserved                      //0x00000014
b   HandlerIRQ  ;handler for IRQ interrupt      //0x00000018
b   HandlerFIQ  ;handler for FIQ interrupt      //0x0000001C
```

3.2 S3C2410X 处理器

本书的驱动大部分是基于三星公司的 ARM9 处理器 S3C2410X。S3C2410X 是一款基于 ARM920T 内核的 RISC 微处理器，主要面向手持式设备以及高性价比、低功耗的应用。S3C2410X 包括如下资源：

（1）集成了大量的功能单元，包括：

- ❑ 内部 1.8V，存储器 3.3V，外部 I/O3.3V，16KB 数据 Cache，16KB 指令 Cache，MMU。
- ❑ 内置外部存储器控制器（SDRAM 控制和芯片选择逻辑）。
- ❑ LCD 控制器，支持 STN 和 TFT 屏。
- ❑ 4 个带外部请求线的 DMA。
- ❑ 3 个通用异步串行端口。
- ❑ 2 通道 SPI 接口。
- ❑ 一个多主 I^2C 总线，一个 I^2S 总线控制器。
- ❑ SD 主接口版本 1.0 和多媒体卡协议版本 2.11 兼容。
- ❑ 两个 USB HOST，一个 USB DEVICE（VER 1.1）。
- ❑ 4 个 PWM 定时器和一个内部定时器。
- ❑ 看门狗定时器。
- ❑ 117 个通用 I/O。
- ❑ 56 个中断源。
- ❑ 24 个外部中断。
- ❑ 电源控制模式：标准、慢速、休眠、掉电。
- ❑ 8 通道 10 位 ADC 和触摸屏接口。
- ❑ 带日历功能的实时时钟。
- ❑ 芯片内置 PLL。
- ❑ 设计用于手持设备和通用嵌入式系统。
- ❑ 16/32 位 RISC 体系结构，使用 ARM920T CPU 核的强大指令集。
- ❑ 带 MMU 的先进的体系结构，支持 WinCE、Linux 等操作系统。
- ❑ ARM920T CPU 核支持 ARM 调试的体系结构。
- ❑ 内部先进的位控制器总线（AMBA）（AMBA2.0，AHB/APB）。

（2）系统管理。

- ❑ 小端/大端支持。
- ❑ 地址空间：一共 8 个存储器 BANK，每个 BANK 128MB，共 1GB。
- ❑ 每个 BANK 可编程为 8/16/32 位数据总线。
- ❑ BANK0 到 BANK6 为固定起始地址。
- ❑ BANK7 为可编程 BANK 起始地址和大小。
- ❑ 前 6 个存储器 BANK 用于 ROM、SRAM 和其他。

- ❑ 两个存储器 BANK 用于 ROM、SRAM 和 SDRAM（同步随机存储器）。
- ❑ 支持等待信号用以扩展总线周期。
- ❑ 支持 SDRAM 掉电模式下的自刷新。
- ❑ 支持 NOR/NAND Flash 启动，通过 OM[1：0]选择。当 OM[1：0]=00 时，从 NAND Flash 启动，当 OM[1：0]=01/10 时，从 NOR Flash 启动。

（3）芯片封装。

- ❑ 272-FBGA 封装。

图 3.4 是 S3C2410X 的结构图。图 3.5 是 S3C2410X 的地址空间。

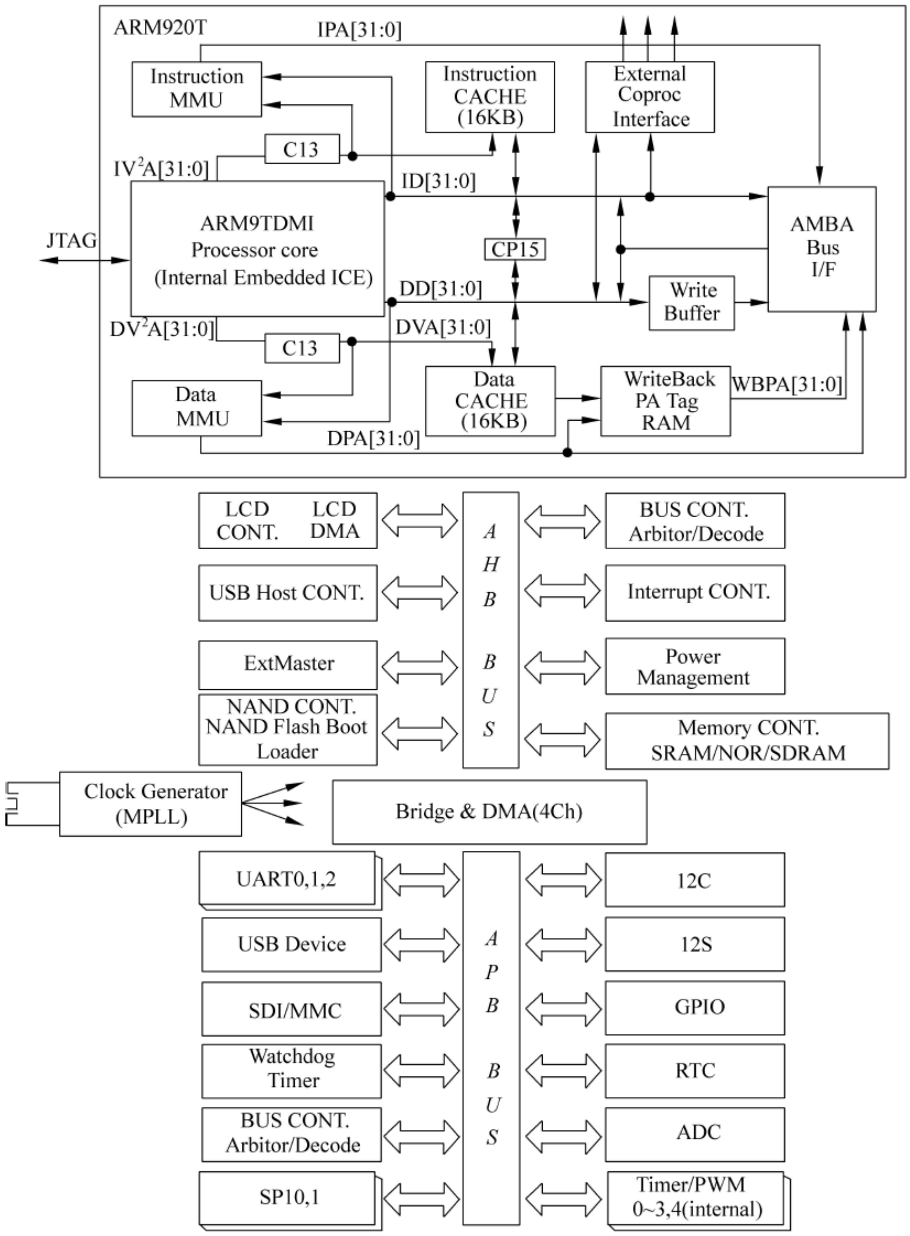


图 3.4　S3C2410X 内部结构图

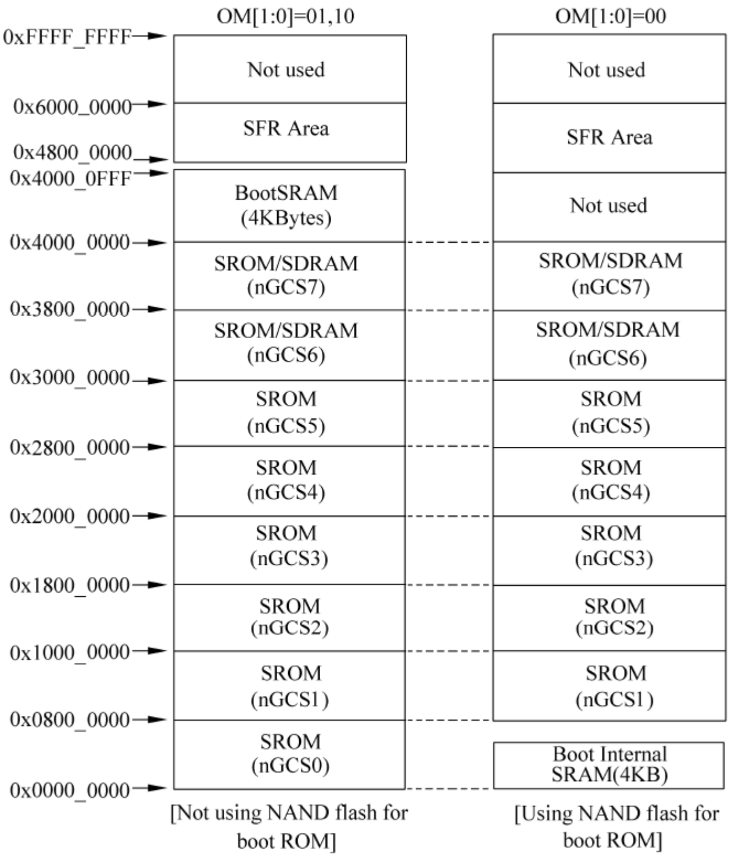


图 3.5　S3C2410X 的地址空间

3.3　S3C2410X I/O 端口

S3C2410X 拥有 117 个多功能的输入输出脚。这些管脚被分成 8 个端口，即 PortA～PortH。其中 PortA 是 23 脚的输出端口，其他端口的管脚均可以设置成输入或输出。每个端口对应一个管脚控制寄存器、一个数据寄存器、一个上拉控制寄存器。管脚控制寄存器主要设置各管脚的功能。数据寄存器用来读取输入数据或向外输出数据。上拉控制寄存器控制管脚上的上拉电阻的使用。本书的驱动例子中用到了端口 F。端口 F 的管脚包括 GPF0～GPF7，关于端口 F 的相关数据如表 3.3～表 3.7 所示。

表 3.3　端口 F 的管脚

Port F	可选的管脚功能			
GPF7	Input/output	EINT7	--	--
GPF6	Input/output	EINT6	--	--
GPF5	Input/output	EINT5	--	--
GPF4	Input/output	EINT4	--	--
GPF3	Input/output	EINT3	--	--
GPF2	Input/output	EINT2		
GPF1	Input/output	EINT1		
GPF0	Input/output	EINT0		

表 3.4　端口 F 的寄存器（包括 GPFCON、GPFDAT、GPFUP）

寄存器	地址	读/写	描述	初始值
GPFCON	0x56000050	读/写	管脚配置寄存器	0x0
GPFDAT	0x56000054	读/写	数据寄存器	不确定
GPFUP	0x56000058	读/写	上拉禁止寄存器	0x0
Reserved	0x5600005C	--	保留	不确定

表 3.5　GPFCON 控制器

GPFCON	位	描述
GPF7	[15:14]	00=输入；01=输出；10=EINT7；11=保留
GPF6	[13:12]	00=输入；01=输出；10=EINT6；11=保留
GPF5	[11:10]	00=输入；01=输出；10=EINT5；11=保留
GPF4	[9:8]	00=输入；01=输出；10=EINT4；11=保留
GPF3	[7:6]	00=输入；01=输出；10=EINT3；11=保留
GPF2	[5:4]	00=输入；01=输出；10=EINT2；11=保留
GPF1	[3:2]	00=输入；01=输出；10=EINT1；11=保留
GPF0	[1:0]	00=输入；01=输出；10=EINT0；11=保留

表 3.6　GPFDAT 数据寄存器

GPFDAT	位	描述
GPF[7:0]	[7:0]	当管脚配置成输入，从这里读取外部数据； 当管脚配置成输出，数据从这里送到管脚； 当管脚配置成功能脚，数据不确定

表 3.7　GPFUP 上拉寄存器

GPFUP	位	描述
GPF[7:0]	[7:0]	0:允许上拉；1:禁止上拉

3.4　最简单的设备驱动——LED 灯驱动

ARM 处理系统中经常使用 GPIO 口驱动 LED 灯作为系统运行状态的指示。采用 S3C2410X 的 GPF4 脚接一个 LED 灯，电路原理图如图 3.6 所示。这个驱动无疑是 ARM 系统中最简单的驱动。下面介绍如何开发这个 LED 灯的驱动。

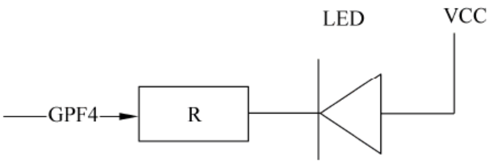


图 3.6　LED 电路图

```
#define LED_SI_OUT
 raw writel(( raw readl(S3C2410 GPFCON)&(~(3<<8)))|(1<<8),S3C2410 GPFCON)
//设置管脚为输出
#define LED_SI_H
```

```
_raw_writel(_raw_readl(S3C2410_GPFDAT)|(1<<4),S3C2410_GPFDAT)
//输出高电平
#define LED_SI_L
 _raw_writel(_raw_readl(S3C2410_GPFDAT)&(~(1<<4)),S3C2410_GPFDAT)
//输出低电平
#define COMMAND_LEDON 1
#define COMMAND_LEDOFF 2
```

利用 ioctl 命令来控制 LED 灯：

```
int DEMO_ioctl(struct inode *inode, struct file *filp,unsigned int cmd, unsigned
long arg)
{
      if(cmd==COMMAND_LEDOFF)
      {
           printk("ioctl LEDOFF successfully\n");
           LED_SI_H;
           return 0;
      }
      if(cmd==COMMAND_LEDON)
      {
           printk("ioctl LEDON successfully\n");
           LED_SI_L;
           return 0;
      }
      printk("ioctl error \n");
      return -EFAULT;
}
```

编译完成后，使用 insmod 加载驱动，然后使用 mknod /dev/led c 224 0 命令建立节点。应用层的测试程序代码如下：

```
main()
{
  int fd;
  int i;
  char data[256];
  int retval;
  fd=open("/dev/led",O_RDWR);
  if(fd==-1)
  {
    perror("error open\n");
    exit(-1);
```

```
    }
    printf("open /dev/led successfully\n");
    retval=ioctl(fd,COMMAND_LEDON,0);
    if(retval==-1)
    {
      perror("ioctl LEDON error\n");
      exit(-1);
    }
    sleep(10);
    retval=ioctl(fd,COMMAND_LEDOFF,0);
    if(retval==-1)
    {
      perror("ioctl LEDOFF error\n");
      exit(-1);
    }
    close(fd);
  }
```

3.5　S3C2410X GPIO 键盘驱动

这里针对 YL2410 开发板上的键盘电路编写一个驱动。如果按照 3.4 节的思路，可以很容易地开发出键盘驱动。键盘电路图如图 3.7 所示。先定义键盘驱动结构：

```
struct DEMO_dev
{
    struct semaphore sem;
    wait_queue_head_t wq;
    struct cdev cdev;
    unsigned char key;//存储最近的按键
};//键盘驱动结构
```

本键盘需要处理 4 个中断：

```
static int irqArray[4]=
{
    IRQ_EINT0,
    IRQ_EINT2,
    IRQ_EINT11,
    IRQ_EINT19
};
```

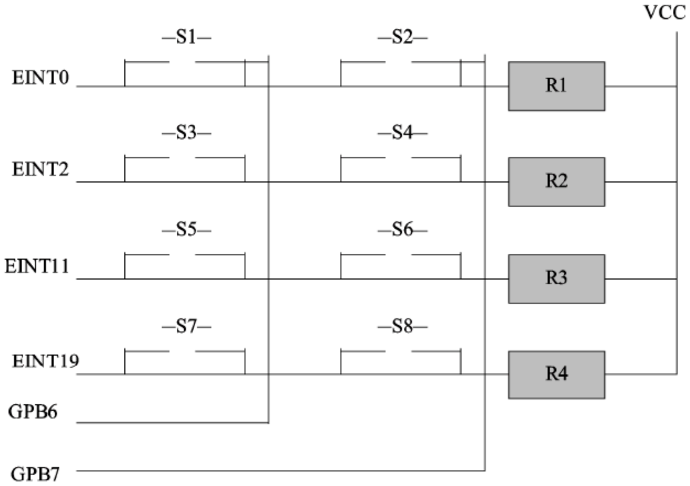


图 3.7　键盘电路图

初始化相关的 I/O 寄存器：

```
void initButton(void)
{
    writel((readl(S3C2410_GPGCON)&(~((3<<22)|(3<<6))))|((2<<22)|(2<<6)),
    S3C24 10_GPGCON);
    writel((readl(S3C2410_GPFCON)&(~((3<<4)|(3<<0))))|((2<<4)|(2<<0)),S3C2410_
    GPFCON) ;
    writel((readl(S3C2410 EXTINT0)&(~(7|(7<<8)))),S3C2410 EXTINT0);
    writel((readl(S3C2410 EXTINT0)|(0|(0<<8))),S3C2410 EXTINT0);
    writel((readl(S3C2410 EXTINT1)&(~(7<<12))),S3C2410 EXTINT1);
    writel((readl(S3C2410 EXTINT1)|(0<<12)),S3C2410 EXTINT1);
    writel((readl(S3C2410 EXTINT2)&(~(0xf<<12))),S3C2410 EXTINT2);
    writel((readl(S3C2410 EXTINT2)|(0<<12)),S3C2410 EXTINT2);
    writel((readl(S3C2410_GPBCON)&(~((3<<12)|(3<<14)))),S3C2410_GPBCON);
    writel((readl(S3C2410_GPBCON)|(((1<<12)|(1<<14)))),S3C2410_GPBCON);
    writel((readl(S3C2410_GPBUP)|(3<<6)),S3C2410_GPBUP);
    writel((readl(S3C2410_GPBDAT)&(~(3<<6))),S3C2410_GPBDAT);
    writel((readl(S3C2410_EINTPEND)|((1<<11)|(1<<19))),S3C2410_EINTPEND);
    writel((readl(S3C2410_EINTMASK)&(~((1<<11)|(1<<19)))),S3C2410_EINTMASK);
}
```

模块初始化的时候，可以申请中断，并初始化定时器和等待队列：

```
for (i = 0; i <4; i++) {
        if (request_irq(irqArray[i], &simplekey_interrupt,
                                                SA_INTERRUPT, "simplekey", NULL)) {
            printk("request button irq failed!\n");
            return -1;
        }
```

```
    }
    init_waitqueue_head(&DEMO_devices->wq);
    init_timer(&polling_timer);
    polling_timer.data = (unsigned long)0;
    polling_timer.function = polling_handler;
```


然后在中断处理中读取键值，保存在 DEMO_devices->key 中：

```
static irqreturn_t simplekey_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    disable_irq(IRQ_EINT0);
    disable_irq(IRQ_EINT2);
    disable_irq(IRQ_EINT11);
    disable_irq(IRQ_EINT19);
    polling_timer.expires = jiffies + HZ/5;
    add_timer(&polling_timer);
    return IRQ_HANDLED;
}
//定时器处理函数
void polling_handler(unsigned long data)
{
    int code=-1;
    writel(readl(S3C2410_SRCPND)&0xFFFFFFDA,S3C2410_SRCPND);
    mdelay(1);
    //扫描按键表，根据中断号，找出所按下的按键
    writel(readl(S3C2410_GPBDAT)|0x80,S3C2410_GPBDAT);//设置 GPB76 为 10
    writel(readl(S3C2410_GPBDAT)&0xFFFFFFBF,S3C2410_GPBDAT);
    if((readl(S3C2410_GPFDAT)&(1<< 0)) == 0 )
    {
        code=0;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPFDAT)&(1<< 2)) == 0 )
    {
        code=2;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPGDAT)&(1<< 3)) ==0 )
    {
        code=4;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPGDAT)&(1<<11)) == 0 )
    {
        code=6;
        goto IRQ_OUT;
```

```
    }
    writel(readl(S3C2410_SRCPND)&0xFFFFFFDA,S3C2410_SRCPND);
    mdelay(1);
    writel(readl(S3C2410_GPBDAT)|0x40,S3C2410_GPBDAT);//设置 GPB76 为 01
    writel(readl(S3C2410_GPBDAT)&0xFFFFFF7F,S3C2410_GPBDAT);
   if((readl(S3C2410_GPFDAT)&(1<< 0)) == 0 )
    {
        code=1;
        goto IRQ OUT;
    }
    else if( (readl(S3C2410 GPFDAT)&(1<< 2)) == 0 )
    {
        code=3;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPGDAT)&(1<< 3)) ==0 )
    {
        code=5;
        goto IRQ OUT;
    }
    else if( (readl(S3C2410 GPGDAT)&(1<<11)) == 0 )
    {
        code=7;
        goto IRQ OUT;
    }
IRQ_OUT:
    enable_irq(IRQ_EINT0);
    enable_irq(IRQ_EINT2);
    enable_irq(IRQ_EINT11);
    enable_irq(IRQ_EINT19);
    if(code>=0)
    {
        //去抖判断
        if((jiffies-polling jffs)>100)
        {
            polling_jffs=jiffies;
            //获取键盘值
            DEMO_devices->key=code;
            printk("get key %d\n",DEMO_devices->key);
            flag = 1;
            wake_up_interruptible(&(DEMO_devices->wq));
        }
    }
    writel(readl(S3C2410 GPBDAT)&0xFFFFFF3F,S3C2410 GPBDAT);
}
```

下面可以通过 read 不断查询键盘的键值：

```
ssize_t DEMO_read(struct file *filp, char _user *buf, size_t count,loff_t *f_pos)
{
    struct DEMO_dev *dev = filp->private_data;
    int sum=0;
    if(flag==1)
    {
        flag = 0;
        sum=1;
        if (copy_to_user(buf,&dev->key,1))
        {
            sum=-EFAULT;
        }
    }
    else
    {
        if (filp->f_flags & O_NONBLOCK)//是否阻塞
        {
            return -EAGAIN;
        }
        else
        {
            if(wait_event_interruptible(dev->wq, flag != 0))
            {
                return-ERESTARTSYS;
            }
            flag = 0;
            sum=1;
            if (copy to user(buf,&dev->key,1))
            {
                sum=-EFAULT;
            }
        }
    }
    return sum;
}
```

最后实现 select 函数接口：

```
unsigned int DEMO_poll(struct file *filp, poll_table *wait)
{
    struct DEMO_dev *dev = filp->private_data;
    poll_wait(filp, &dev->wq, wait);
    if (flag==1)//数据准备好
        return  POLLIN | POLLRDNORM;
```

```
    return 0;
}
```

测试程序代码如下：

```
int main(void)
{
  int buttons fd;
  unsigned char key value;
  buttons fd = open("/dev/buttons", 0);
  if (buttons fd < 0) {
    perror("open device buttons");
    exit(1);
  }
  while(1)
  {
    fd set rds;
    int ret;
    FD_ZERO(&rds);
    FD_SET(buttons_fd, &rds);
    ret = select(buttons_fd + 1, &rds, NULL, NULL, NULL);
    if (ret < 0) {
        perror("select");
        exit(1);
    }
    if (ret == 0)
    {
        printf("select timeout.\n");
    }
    else if (FD ISSET(buttons fd, &rds))
     {
        int ret = read(buttons fd, &key value, sizeof key value);
        if (ret != sizeof key_value)
        {
            if (errno != EAGAIN)
              perror("read buttons\n");
            continue;
        }
     else
        {
            printf("get buttons_value: %d\n", key_value);
        }
    }
  }
  close(buttons fd);
  return 0;
}
```

测试结果如下：

```
[root@(none) tmp]# insmod demo.ko
Using demo.ko
[root@(none) tmp]# mknod /dev/buttons c 224 0
[root@(none) tmp]# ./read
get key 4
get buttons_value: 4
get key 0
get buttons_value: 0
```

第 4 章 串行总线驱动

串行总线包括 I^2C、SMBUS、SPI、CAN、RS232、USB 等。在 Linux 下开发串行总线驱动的最简单的方法就是利用 GPIO 口实现总线时序。有的处理器提供了相关的总线控制器，则不需要用户自己管理总线时序。另外新的 Linux 内核也对部分总线驱动进行了抽象，比如 I^2C 总线和 USB 总线。本章重点介绍 I^2C 和 SPI 驱动开发，下一章介绍 USB 总线驱动开发。

4.1 串行总线综述

在嵌入式系统中，越来越多的处理器和控制器用不同类型的总线集成在一起，它们之间目前流行的通信一般采用串行或并行模式，而串行模式应用更广泛。串行相对于并行的主要优点是要求的引脚数较少。集成在一个微控制器中的并行总线一般需要 8 条或更多的引脚，引脚数的多少取决于设计中地址和数据的宽度，所以集成一个并行总线的芯片至少需要 8 个引脚来与外部器件接口，这增加了芯片的总体尺寸。相反地，使用串行总线可以将同样的芯片集成在一个较小的封装中。

微处理器中常用的集成串行总线是通用异步接收器传输总线（UART）、串行通信接口（SCI）、同步外设接口（SPI）、内部集成电路（I^2C）和通用串行总线（USB），以及车用串行总线，包括控制器区域网（CAN）和本地互联网（LIN）。这些总线在速度、物理接口要求和通信方法上都有所不同。

4.1.1 I^2C 总线

I^2C（Inter-Integrated Circuit）总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。I^2C 总线产生于 20 世纪 80 年代，最初为音频和视频设备开发，如今主要在服务器管理中使用，其中包括单个组件状态的通信。目前有很多单片机、ARM 处理器，以及外围器件，如存储器、监控芯片等都提供 I^2C 接口。串行的 8 位双向数据传输位速率在标准模式下可达 100Kbps，快速模式下可达 400Kbps，高速模式下可达 3.4Mbps。

I^2C 总线最主要的优点是其简单性和有效性。由于接口直接在组件之上，因此 I^2C 总线占用的空间非常小，减少了电路板的空间和芯片管脚的数量，降低了互联成本。总线的长度可高达 25 英尺，并且能够以 10Kbps 的最大传输速率支持 40 个组件。I^2C 总线的另一个优点是它支持多主控制，其中任何能够进行发送和接收的设备都可以成为主总线。I^2C 总线在任何时间点上只能有一个主控，主控能够控制信号的传输和时钟频率。

I^2C 总线是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。每个器件都有一个唯一的地址识别。发送数据到总线的器件叫发送器，从总线接收数据的器件叫接收器。初始化发送产生时钟信号和终止发送的器件叫主机，被主机寻址的器件叫从机。

I^2C 总线是一个多主机的总线。当有多于一个主机尝试控制总线时，通过仲裁只允许其中一个控制总线并使报文不被破坏。仲裁的原则是：当多个主器件同时想占用总线时，如果某个主器件发送高电平，而另一个主器件发送低电平，则发送电平与此时 SDA 总线电平不符的那个器件将自动关闭其输出级。总线竞争的仲裁是在两个层次上进行的。首先是地址位的比较，如果主器件寻址同一个从器件，则进入数据位的比较，从而确保了竞争仲裁的可靠性。由于是利用 I^2C 总线上的信息进行仲裁，因此不会造成信息的丢失。

1. 数据的有效性

SDA 线上的数据必须在时钟的高电平周期保持稳定，数据线的高或低电平状态只有在 SCL 线的时钟信号是低电平时才能改变，如图 4.1 所示。

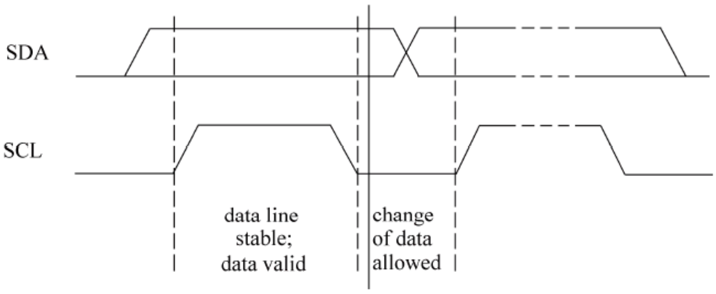


图 4.1 I^2C 数据有效性

2. 起始和停止条件

在 I^2C 总线中唯一出现的是被定义为起始 S 和停止 P 条件的情况。其中一种情况是在 SCL 线是高电平时，SDA 线从高电平向低电平切换，这个情况表示起始条件。当 SCL 线是高电平时，SDA 线由低电平向高电平切换，表示停止条件。起始和停止条件一般由主机产生。总线在起始条件后被认为处于忙的状态，在停止条件的某段时间后总线被认为再次处于空闲状态。如图 4.2 所示。

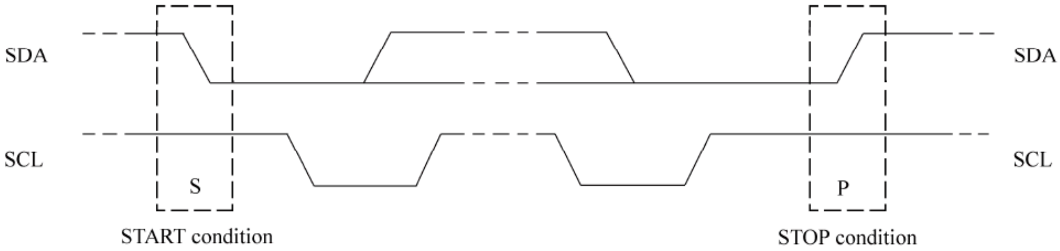


图 4.2 起始和停止条件

3. 传输数据

发送到 SDA 线上的每个字节必须为 8 位。每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位。首先传输的是数据的最高位 MSB。直到从机完成一些其他的功能后，例如一个内部中断服务程序，才能接收或发送下一个完整的数据字节。可以使时钟线 SCL 保持低电平，迫使主机进入等待状态，当从机准备好接收下一个数据字节，并释放时钟线 SCL 后，数据传输继续。数据传输必须带响应。相关的响应时钟脉冲由主机产生。在响应的时钟脉冲期间发送

器释放 SDA 线。在响应的时钟脉冲期间，接收器必须将 SDA 线拉低，使它在这个时钟脉冲的高电平期间保持稳定的低电平，I²C 数据传送时序如图 4.3 所示。

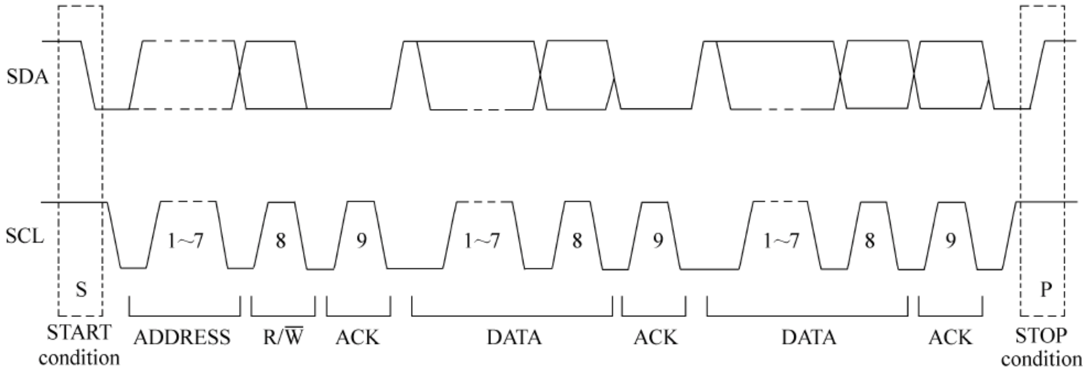


图 4.3　I²C 数据传送时序

如图 4.4、图 4.5 所示是一种 7 位寻址的 I²C 读写的实例。其中阴影部分是主机发向从机，白色部分是从机发向主机。A 代表响应，Ā 表示无应答，S 表示开始，P 表示停止。

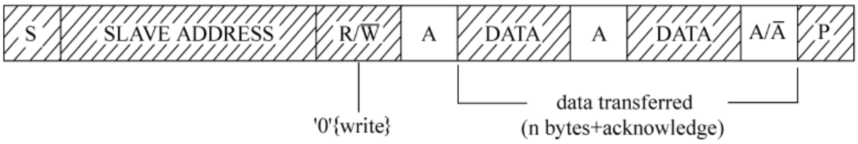


图 4.4　I²C 数据写操作

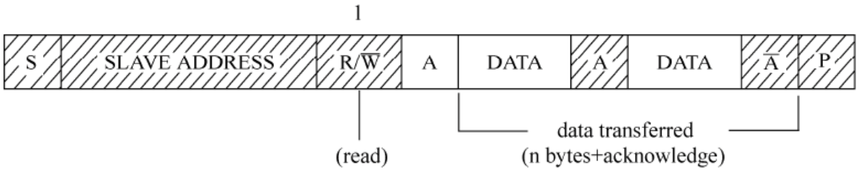


图 4.5　I²C 数据读操作

4.1.2　SMBus 总线

SMBus 是 System Management Bus 的缩写，是 1995 年由 Intel 提出的，应用于移动 PC 和桌面 PC 系统中的低速率通信。它主要是希望通过一条廉价并且功能强大的总线（由两条线组成）来控制主板上的设备并收集相应的信息。SMBus 为系统和电源管理这样的任务提供了一条控制总线，使用 SMBus 的系统，设备之间发送和接收消息都是通过 SMBus，而不是使用单独的控制线，这样可以节省设备的管脚数。使用 SMBus，设备还可以提供它的生产信息，告诉系统它的型号，部件号等，针对一些事件保存它的状态，报告不同类别的错误，接收控制参数，并返回它的状态等。SMBus 最适用于笔记本电脑，检测各元件状态并更新硬件设置引脚（pull-high 或 pull-low）。例如，将不存在的 DIMM 时钟关闭，或检测电池低电压状态。SMBus 的数据传输率只有 100Kbps；

这允许单一主机与 CPU 和多个主从硬盘通信并收发数据。SMBus 也可用于免跳线设计的主板上。

SMBus 也是一种二线制串行总线，它大部分基于于 I^2C 总线规范，但只工作在 100kHz，且专门面向智能电池管理应用。它工作在主/从模式：主器件提供时钟，在其发起一次传输时提供一个起始位，在其终止一次传输时提供一个停止位；从器件拥有一个唯一的 7 或 10 位从器件地址。SMBus 与 I^2C 总线之间在时序特性上存在一些差别。首先，SMBus 需要一定数据保持时间，而 I^2C 总线则是从内部延长数据保持时间。SMBus 具有超时功能，因此当 SCL 线太低而超过 35 ms 时，从器件将复位正在进行的通信。相反，I^2C 采用硬件复位。SMBus 具有一种警报响应地址（ARA），因此当从器件产生一个中断时，它不会马上清除中断，而是一直保持到其收到一个由主器件发送的含有其地址的 ARA 为止。I^2C 具有 400kHz 与 2MHz 两个版本，SMBus 只工作在从 10～100kHz。最低工作频率 10kHz 是由 SMBus 超时功能决定的。

4.1.3 SPI 总线

SPI 总线技术是 Motrrola 公司推出的一种同步串行接口。它是一种四线制串行总线接口，为主/从结构，4 条导线分别为串行时钟（SCLK）、主出从入（MOSI）、主入从出（MISO）和从选（SS）信号：

（1）MOSI：主器件数据输出，从器件数据输入。

（2）MISO：主器件数据输入，从器件数据输出。

（3）SCLK：时钟信号，由主器件产生。

（4）SS：从器件使能信号，由主器件控制。

其中，从选择线只用于从属模式。主器件为时钟提供者，可发起读从器件或写从器件操作。这时主器件将与一个从器件进行对话。当总线上存在多个从器件时，要发起一次传输，主器件将把该从器件选择线拉低，然后分别通过 MOSI 和 MISO 线启动数据发送或接收。SPI 时钟速度很快，范围可从几兆赫兹到几十兆赫兹，且没有系统开销。SPI 在系统管理方面的缺点是缺乏流控机制，无论主器件还是从器件均不对消息进行确认，主器件无法知道从器件是否繁忙。

SPI 接口是以主从方式工作的，这种模式通常有一个主器件和一个或多个从器件，如图 4.6 所示。在点对点的通信中，SPI 接口不需要进行寻址操作，且为全双工通信，显得简单高效。在多个从器件的系统中，每个从器件需要独立的使能信号，硬件上比 I^2C 系统要稍微复杂一些。

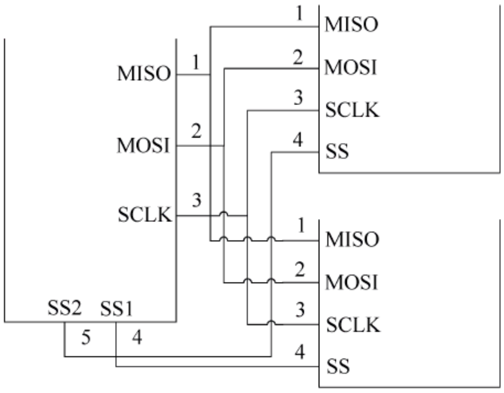


图 4.6　一主多从的 SPI

4.1.4 CAN 总线

控制器区域网（Controller Area Network，CAN）现场总线已经成为在仪表装置通信的新标准。它提供高速数据传送，在短距离（40m）条件下具有高速（1Mbps）数据传输能力，而在最大距

离 10 000m 时具有低速（5Kbps）传输能力，极适合在高速的工业自控应用上。CAN 总线可在同一网络上连接多种不同功用的传感器（如位置，温度或压力等）。

CAN 总线是一种多主总线，即每个节点机均可成为主机，且节点机之间也可进行通信，CAN 的通信介质可以是双绞线、同轴电缆或光导纤维。CAN 总线通信接口中集成了 CAN 协议的物理层和数据链路层功能，可完成对通信数据的成帧处理，包括位填充、数据块编码、循环冗余校验、优先级判别等项工作。CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块进行编码。采用这种方法的优点是可使网络内的节点个数在理论上不受限制，数据块的标识码可由 11 位或 29 位二进制数组成，因此可以定义 211 或 229 个不同的数据块，这种按数据块编码的方式还可使不同的节点同时接收到相同的数据，这一点在分步式控制中非常重要。CAN 数据段长度最多为 8 个字节，可满足通常工业领域中控制命令，工作状态及测试数据的一般要求。同时，8 个字节不会占用总线时间过长，从而保证了通信的实时性。CAN 协议采用 CRC 检验，并可提供相应的错误处理功能，保证了数据通信的可靠性。CAN 总线节点在错误严重的情况下具有自动关闭输出功能，以使总线上其他节点的操作不受影响。

CAN 总线使用一种叫做“载波监测，多主掌控/冲突避免”（CSMA/CA）的通信模式。如果同一时刻有两个以上的设备想发送信息，CAN 总线能够实时监测这些冲突，并做出仲裁，而使获得仲裁的信息帧不受任何损坏地继续传送。CAN 总线解决总线竞争的方法是按位对标识符进行仲裁，各发送节点在向总线发送电平的同时，也对总线上的电平进行读取，并与自身发送的电平进行比较，如果电平相同则继续发送下一位，否则停止发送，退出竞争。

CAN 报文传输由以下 4 个不同的帧类型表示和控制。

（1）数据帧：数据帧携带数据从发送器至接收器。

（2）远程帧：总线单元发出远程帧，请求发送具有同一识别符的数据帧。

（3）错误帧：任何单元检测到一总线错误就发出错误帧。

（4）过载帧：过载帧用以在先行的和后续的数据帧（或远程帧）之间提供一附加的延时。

CAN 帧格式包括标准格式和扩展格式。标准 CAN 的标志符长度是 11 位，而扩展格式 CAN 的标志符长度可达 29 位。CAN 2.0A 版本规定 CAN 控制器必须有一个 11 位的标志符。同时，在 CAN 2.0B 版本中规定，CAN 控制器的标志符长度可以是 11 位或 29 位。遵循 CAN 2.0B 协议的 CAN 控制器可以发送和接收 11 位标识符的标准格式报文或 29 位标识符的扩展格式报文。如果禁止 CAN 2.0B，则 CAN 控制器只能发送和接收 11 位标识符的标准格式报文，而忽略扩展格式的报文结构，但不会出现错误。

下面是 CAN 2.0B 协议帧格式。

1. CAN 2.0B 标准帧

CAN 标准帧信息为 11 个字节，包括两部分：信息和数据部分。前三个字节为信息部分，如表 4.1 所示。

表 4.1 CAN 2.0B 标准帧格式

	7	6	5	4	3	2	1	0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码） ID.10～ID.3							
字节 3	ID.2～ID.0			X	X	X	X	X

续表

	7	6	5	4	3	2	1	0
字节 4	数据 1							
字节 5	数据 2							
字节 6	数据 3							
字节 7	数据 4							
字节 8	数据 5							
字节 9	数据 6							
字节 10	数据 7							
字节 11	数据 8							

字节 1 为帧信息。第 7 位（FF）表示帧格式，在标准帧中，FF＝0；第 6 位（RTR）表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示在数据帧时实际的数据长度。

字节 2、3 为报文识别码，只有 11 位有效。

字节 4～11 为数据帧的实际数据，远程帧时无效。

2. CAN 2.0B 扩展帧

CAN 扩展帧信息为 13 个字节，包括两部分，信息和数据部分。前 5 个字节为信息部分，如表 4.2 所示。

表 4.2　CAN 2.0B 扩展帧格式

	7	6	5	4	3	2	1	0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码）ID.28～ID.21							
字节 3	ID.20～ID.13							
字节 4	ID.12～ID.5							
字节 5	ID.4～ID.0					X	X	X
字节 6	数据 1							
字节 7	数据 2							
字节 8	数据 3							
字节 9	数据 4							
字节 10	数据 5							
字节 11	数据 6							
字节 12	数据 7							
字节 13	数据 8							

（1）字节 1 为帧信息。第 7 位（FF）表示帧格式，在扩展帧中，FF=1；第 6 位（RTR）表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示在数据帧时实际的数据长度。

（2）字节 2～5 为报文识别码，其高 29 位有效。

（3）字节 6～13 为数据帧的实际数据，远程帧时无效。

在现有的底层协议（物理层和数据链路层）之上可以有更高层的协议，这就是 CAN 的高层协议。CAN 的高层协议也可理解为应用层协议，它是一种高层协议，是在 CAN 规范的基础上发

展起来的应用层。许多系统中，可以特别制定一个合适的应用层，但对于许多的行业来说，这种方法是不经济的。一些组织已经研究并开放了应用层标准，以使系统的综合应用变得十分容易。一些常见的 CAN 高层协议有：CAL 协议、CANOpen 协议、DeviceNet 协议、SDS 协议、CANKingdom 协议等。

4.2 CAN 接口芯片 MCP2510

MCP2510 是 Microchip 公司推出的功能很强的 CAN 控制器芯片，它支持 CAN 1.2、CAN 2.0A 及 CAN 2.0B 规范；其内部结构见图 4.7 所示。该芯片内含三个发送缓存和两个接收缓存，可以对发送优先级进行管理，可滤除无用信息，MCP2510 有 6 个可编程滤波器，而且中断资源十分丰富。最可贵的是，它可以通过标准的 SPI 接口与微控制器进行通信，从而放宽了 MCU 的选择范围，使得所有单片机都有接入的可能。MCP2510 的主要功能是在 MCU 的控制下实现 CAN 规范，它内部的所有寄存器和控制寄存器都映射到一个地址表上，MCU 可以使用相应的命令格式通过标准的 SPI 接口来完成对 MCP2510 的初始化、工作状态的控制以及对数据的读写。此外，MCP2510 产生的中断还可以反馈给 MCU 来处理。

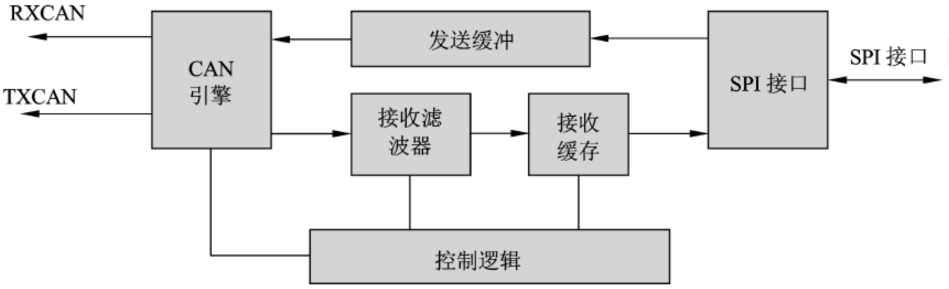


图 4.7 MCP2510 的结构图

4.2.1 数据发送

MCP2510 采用三个发送缓冲器。每个发送缓冲器占据 14 个字节的 SRAM，并映射到存储器中。其中第一字节 TXBNCTRL 是与报文缓冲器相关的控制寄存器。该寄存器中的信息决定了报文在何种条件下被发送，并在报文发送时指示其状态。接着的 5 个字节用来装载标准和扩展标识符以及其他报文仲裁信息。最后 8 个字节用来装载等待发送的报文的 8 个可能的数据字节，通过设定控制寄存器中 TXBNCTRL.TXREQ 发送控制位可以启动相应发送缓冲器的报文发送。通过 SPI 接口写寄存器或向某一发送缓冲器的 TXNRTS 引脚输入低电平可以进行设定。如果选择 SPI 接口方式进行位设定以启动报文发送，可以同时设定 TXREQ 位和 TXP 优先级控制位。下面介绍发送缓冲寄存器。

（1）发送缓冲器 N 的标准标识符高位，如图 4.8 所示。

```
SID<10:3>: 标准标识符 <10:3>。
```

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit7							bit 0

图 4.8　发送缓冲器 N 的标准标识符高位

（2）发送缓冲器 N 的标准标识符低位，如图 4.9 所示。

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

图 4.9　发送缓冲器 N 的标准标识符低位

```
SID<2:0>: 标准标识符位数 <2:0>。
bit 4 未用: 读作 “0” 。
EXIDE: 扩展标识符使能。
1 = 报文将发送扩展标识符;
0 = 报文将发送标准标识符;
bit 2 未用: 读作 “0” 。
EID<17:16>: 扩展标识符位数 s <17:16>。
```

（3）发送缓冲器 N 扩展标识符高位，如图 4.10 所示。

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8
bit 7							bit 0

图 4.10　发送缓冲器 N 扩展标识符高位

```
EID<15:8>: 扩展标识符位数 <15:8>。
```

（4）发送缓冲器 N 扩展标识符低位，如图 4.11 所示。

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
EID7	EID6	EID5	EID4	EID3	EID2	EID1	EID0
bit 7							bit 0

图 4.11　发送缓冲器 N 扩展标识符低位

```
EID<7:0>: 扩展标识符位数 <7:0>。
```

（5）发送缓冲器 N 数据长度码，如图 4.12 所示。

R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	RTR	—	—	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

图 4.12　发送缓冲器 N 数据长度码

```
bit 7 未用: 读作 “0” 。
```

```
RTR：远程发送请求位。
1 = 发送的报文为远程帧；
0 = 发送的报文为数据帧。
bit 5、4 未用：读作“0”。
DLC<3:0>：数据长度码。设定发送的数据长度（0～8 字节）。
```

设 can_id 为数据包的 ID。填写发送报文标识符寄存器的方法如下：

```
void MCP2510_Write_ID(unsigned char address, U32 can_id, int IsExt)
{
    unsigned char buffer[4];
    memset(buffer,0,4);
    if(IsExt)//扩展帧
    {
        U32 stdId=can_id&(0x3FF<<18);
        U32 ExtID=can_id&0x3FFFF;
        buffer[0]=stdId>>3;
        buffer[1]=(stdId&0x7)<<5;
        buffer[1]|=0x08;//扩展标识
        buffer[1]|=ExtID>>16;
        buffer[2]=(ExtID&0xFF00)>>8;
        buffer[3]=ExtID&0xFF;
    }
    else//标准帧
    {
        buffer[0]=can_id>>3;
        buffer[1]=(can_id&0x7)<<5;
    }
    MCP2510_Swrite(address, (unsigned char*)&buffer, 4);
}
```

MCP2510_Swrite 将 buffer 写到发送缓冲器 N 的标准标识符高位寄存器地址开始的 4 字节。

4.2.2 数据接收

MCP2510 具有两个全文接收缓冲器。每个接收缓冲器配备有多个验收滤波器。除上述的专用接收缓冲器外，MCP2510 还具有单独的报文集成缓冲器（MAB），可作为第三个接收缓冲器。在三个接收缓冲器中，MAB 总能够接收来自总线的下一条报文。其余两个接收缓冲器 RXB0 和 RXB1 则从协议引擎接收完整的报文。当其中一个缓冲器处于接收等待或保存着上一条接收到的报文时，MCU 可对另一缓冲器进行访问。MAB 对接收到的报文进行组合，并将满足验收滤波器条件的报文传送到 RXBN 缓冲器。当报文传送至某一接收缓冲器，与该接收缓冲器对应的 CANINTF.RXNIF 位将置 1。一旦缓冲器中的报文处理完毕，MCU 就必须将该位清除以接收下一条报文。该控制位提供的锁定功能确保在 MCU 尚未处理完上一条报文前，MCP2510 不会将新的报文载入接收缓冲器。如果 CANINTE 的 RXNIE 位被置 1，器件会在 INT 引脚产生一个中断，显

示接收到的有效报文。

可以通过设置接收缓冲器控制寄存器的 RXM<1:0>来选择接收扩展帧还是标准帧。图 4.13 是接收缓冲器 0 控制寄存器 RXB0CTRL。

U-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	BUKT	BUKT1	FILHIT0
bit 7							bit 0

图 4.13　接收缓冲器 0 控制器 RXB0CTRL

RXM<1:0>: 接收缓冲器工作模式。
11=关闭屏蔽/滤波功能; 接收所有报文;
10=只接收符合滤波器条件的带扩展标识符的有效报文;
01=只接收符合滤波器条件的带标准标识符的有效报文;
00=接收符合滤波器条件的所有带扩展标识符或标准标识符的有效报文。
RXRTR: 是否收到远程传递请求。
1=接收到远程传递请求;
0=未收到远程传递请求。
BUKT: 滚存使能。
1=如果 RXB0 满，RXB0 接收到的报文将被滚存至 RXB1;
0=滚存禁止。
Bit 1 BUKT1: 只读位，BUKT 位备份（只在 MCP2510 器件内部使用）。
Bit 0 FILHIT<0>: 滤波器指示——指明使能报文接收的验收滤波寄存器编号。
1=验收滤波寄存器 1（RXF1）;
0=验收滤波寄存器 0（RXF0）。
注:如果从 RXB0 到 RXB1 的滚存发生，FILHIT 位将反映接收滚存报文的滤波器。

图 4.14 是接收缓冲器 1 控制寄存器 RXB1CTRL。

U-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0	R-0
—	RXM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0
bit 7							bit 0

图 4.14　接收缓冲器 1 控制寄存器 RXB1CTRL

RXM<1:0>: 接收缓冲器工作模式。
11=关闭屏蔽/滤波功能; 接收任何报文;
10=只接收符合滤波器条件的带扩展标识符的有效报文;
01=只接收符合滤波器条件的带标准标识符的有效报文;
00=接收符合滤波器条件的所有带扩展标识符或标准标识符的有效报文。
当新报文符合验收滤波条件并被载入接收缓冲器时，使能报文接收的滤波器编号将被装载到 RXBNCTRL 寄存器 FILHIT 位中。
FILHIT<2:0>: 滤波器指示——显示使能报文接收的过滤寄存器编号。
101=验收滤波寄存器 5（RXF5）;
100=验收滤波寄存器 4（RXF4）;
011=验收滤波寄存器 3（RXF3）;
010=验收滤波寄存器 2（RXF2）;

```
001=验收滤波寄存器 1（RXF1）（只有当 RXB0CTRL 中的 BUKT 位置 1 时）；
000=验收滤波寄存器 0（RXF0）（只有当 RXB0CTRL 中的 BUKT 位置 1 时）。
```

MCP2510 的接收滤波器共有 6 组，即 RXFn（n=0～5）。每个滤波器包括 4 个寄存器。

（1）RXFnSIDH：验收滤波寄存器 N 标准标识符的高位。

（2）RXFnSIDL：验收滤波寄存器 N 标准标识符的低位。

（3）RXFnEID8：验收滤波器 N 扩展标识符的高位。

（4）RXFnEID0：验收滤波寄存器 N 扩展标识符的低位。

对应的屏蔽寄存器也有 6 组，即 RXMn（n=0～5）。

（1）RXMnSIDH：验收滤波屏蔽寄存器 N 标准标识符的高位。

（2）RXMnSIDL：验收滤波屏蔽寄存器 N 标准标识符的低位。

（3）RXMnEID8：验收滤波屏蔽寄存器 N 扩展标识符的高位。

（4）RXMnEID0：验收滤波屏蔽寄存器 N 扩展标识符的低位。

表 4.3 是 MCP2510 的滤波算法真值表。当屏蔽位为 0 时，接收任何值；当屏蔽位为 1 时，只接收滤波器对应位设置的值。

表 4.3　滤波/屏蔽寄存器真值表

屏蔽位 n	过滤位 n	报文标识符位 n001	接收/拒绝位 n
0	X	X	接收
1	0	0	接收
1	0	1	拒绝
1	1	0	拒绝
1	1	1	接收

注意：X=任意值。

4.2.3　中断

MCP2510 具有 8 个中断源。CANINTE 寄存器中包含了使能各个中断源的中断使能控制位。CANINTF 寄存器中包含了各个中断源的中断标志位。当有中断请求发生时，INT 引脚将置为低电平，并维持低电平状态直至 MCU 清除中断标志。中断标志只有在引起相应中断请求条件消失后，才能被清除。建议在对 CANINTF 寄存器中的中断标志位进行复位操作时，采用位修改命令而不要使用普通的写操作。这是为了避免在写命令执行中无意间修改了标志位，从而导致中断请求信号的丢失。应注意，CANINTF 中的中断标志位为可读写位，因此在相关 CANINTE 中断使能位置 1 的前提下，对上述任何一位进行置位均可使 MCU 产生中断请求。

（1）中断使能寄存器，如图 4.15 所示。

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE
bit 7							bit 0

图 4.15　中断使寄存器

```
bit 7 MERRE: 报文错误中断使能。
```

```
bit 6 WAKIE: 唤醒中断使能。
bit 5 ERRIE: 错误中断使能。
bit 4 TX2IE: 发送缓冲器 2 空中断使能。
bit 3 TX1IE: 发送缓冲器 1 空中断使能。
bit 2 TX0IE: 发送缓冲器 0 空中断使能。
bit 1 RX1IE: 接收缓冲器 1 满中断使能。
bit 0 RX0IE: 接收缓冲器 0 满中断使能。
```

（2）中断标志寄存器，如图 4.16 所示。

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF
bit 7							bit 0

图 4.16　中断标志寄存器

```
bit 7 MERRF: 报文出错中断标志。
bit 6 WAKIF: 唤醒中断标志。
bit 5 ERRIF: 出错中断标志。
bit 4 TX2IF: 发送缓冲器 2 空中断标志。
bit 3 TX1IF: 发送缓冲器 1 空中断标志。
bit 2 TX0IF: 发送缓冲器 0 空中断标志。
bit 1 RX1IF: 接收缓冲器 1 满中断标志。
bit 0 RX0IF: 接收缓冲器 0 满中断标志。
```

4.2.4　波特率设置

MCP2510 的波特率设置主要与下面的寄存器相关。

（1）配置寄存器 1（CNF1），如图 4.17 所示。

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
bit 7							bit 0

图 4.17　配置寄存器 1

```
bit 7、6 SJW<1:0>: 同步跳转宽度。
11 = 长度 = 4xTQ;
10 = 长度 = 3xTQ;
01 = 长度 = 2xTQ;
00 = 长度 = 1xTQ。
bit 5～0 BRP<5:0>: 波特率预分频器。
TQ = 2×(BRP + 1)/FOSC。
```

（2）配置寄存器 2（CNF2），如图 4.18 所示。

```
bit 7 BTLMODE: 相位段 2 位时间长度。
1 = 相位段 2 位时间长度由 CNF3 中的 PHSEG22:PHSEG20 确定;
```

```
0 = 相位段 2 位时间长度取相位缓冲段 1 和 IPT (2TQ) 之间的较大值。
bit 6 SAM: 采样点配置。
1 = 在采样点对总线进行三次采样;
0 = 在采样点对总线进行一次采样。
bit 5~3 PHSEG1<2:0>: 相位段 1 位时间长度。
(PHSEG1 + 1) × TQ。
bit 2~0 PRSEG<2:0>: 传播段长度。
(PRSEG + 1) × TQ。
```

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0
bit 7							bit 0

图 4.18 配置寄存器 2

（3）配置寄存器 3（CNF3），如图 4.19 所示。

U-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0
—	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20
bit 7							bit 0

图 4.19 配置寄存器 3

```
bit 7 未用: 读作 "0"。
bit 6 WAKFIL: 唤醒输入引脚滤波使能位。
1 = 唤醒输入引脚滤波使能;
0 = 唤醒输入引脚滤波禁止。
bit 5~3 未用: 读作 "0"。
bit 2~0 PHSEG2<2:0>: 相位段 2 长度。
(PHSEG2 + 1) × TQ。
注: 相位段 2 的最小有效设定为 2TQ
```

下面是一个计算波特率的实例：

如果 CNF1=0x93，CNF2=0xAE，CNF3=0x05；晶振是 16MHz，可以计算出 20 分频的时候，一个 TQ 是（2*20）/16000=2.5μs，那么 PS2=6，PS1=6，PROG=7，SYN=1，加起来是 20，20×2.5μs=50μs，波特率就是 20Kbps 了。

4.2.5 工作模式

MCP2510 具有 5 种工作模式，分别如下所示。

（1）配置模式：正常运行之前，必须对 MCP2510 进行初始化。只有在配置模式下，才能对器件进行初始化。

（2）正常模式：该模式下，器件主动监视总线上的所有报文，并产生确认位和错误帧等。只有在正常工作模式下，MCP2510 才能在 CAN 总线上进行报文的传输。

（3）睡眠模式：MCP2510 处于休眠模式，SPI 接口仍能保持正常工作，以允许访问器件内的

所有寄存器。

（4）监听模式：监听模式使 MCP2510 可以接收包括错误报文在内的所有报文。监听模式是一种静音模式，即器件不能发送包括错误标志或确认信号在内的任何报文。

（5）环回模式：该模式可使器件内部发送缓冲器和接收缓冲器之间进行报文自发自收，而无须通过 CAN 总线。该模式可用于系统研发和测试。

通过设定 CANCTRL 的 REQOP 位，可选择工作模式。改变工作模式时，新的工作模式需等到所有报文传输完毕之后才能生效。因此在运行另一种模式之前，用户在进行下一步操作时应先确认器件是否已进入该工作模式。通过读取 CANSTAT 的 OPMODE 位可以查验当前工作模式。

4.3 MCP2510 驱动开发

MCP2510 与 S3C2410X 之间其实是 SPI 接口，开发 MCP2510 的驱动，实际上就是实现 SPI 的时序。图 4.20 是 YL2410 开发板上的 MCP2510 与 S3C2410X 连接的电路原理。

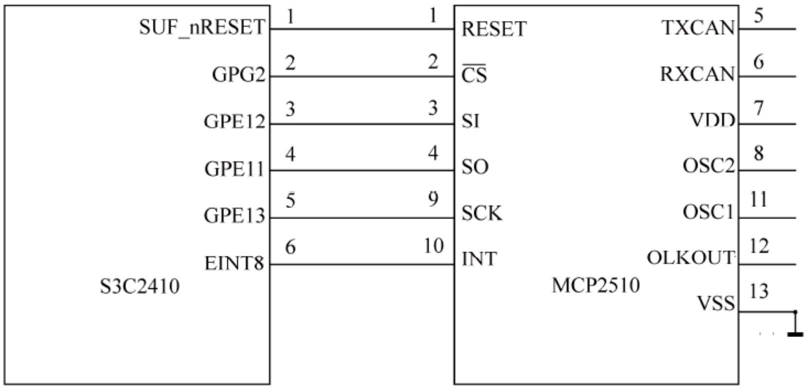


图 4.20　MCP2510 接口电路

下面介绍 MCP2510 SPI 的接口时序。

首先介绍读指令。在读操作开始时，$\overline{CS}$ 引脚将被置为低电平。随后读指令和 8 位地址码（A7～A0）将被依次送入 MCP2510。在接收到读指令和地址码之后，MCP2510 指定地址寄存器中的数据将被移出通过 SO 引脚进行发送。每一数据字节移出后，器件内部的地址指针将自动加一以指向下一地址。因此可以对下一个连续地址寄存器进行读操作。通过该方法可以顺序读取任意个连续地址寄存器中的数据。通过拉高 $\overline{CS}$ 引脚电平可以结束读操作，如图 4.21 所示。

发送写指令时，置 $\overline{CS}$ 引脚为低电平启动写操作。启动写指令后，地址码以及至少一个字节的数据被依次发送到 MCP2510。只要 $\overline{CS}$ 保持低电平，就可以对连续地址寄存器进行顺序写操作。在 SCK 引线上的上升沿，数据字节将从 D0 位开始依次被写入。如果 $\overline{CS}$ 引脚在字节的 8 位数据尚未发送完之前跳变到高电平，该字节的写操作将被中止，而之前发送的字节已经写入。有关详细的字节写操作时序请参见图 4.22。

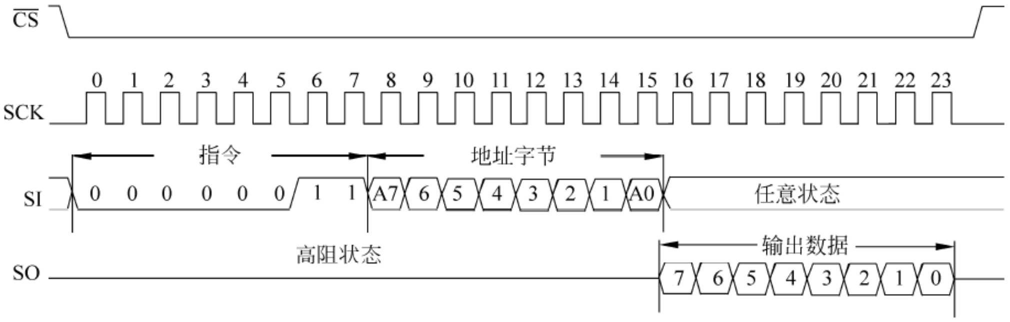


图 4.21　MCP2510 SPI 读

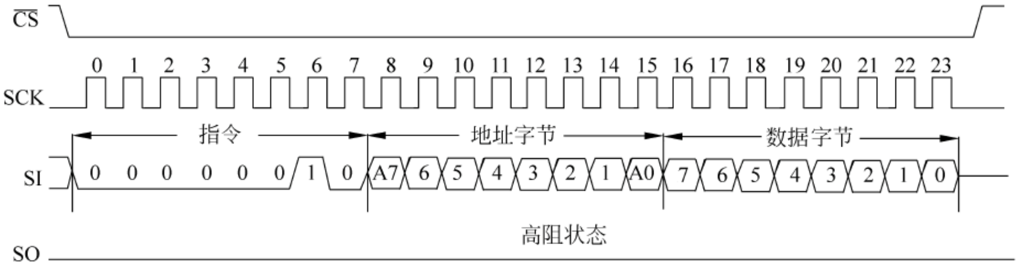


图 4.22　MCP2510 SPI 写

首先定义几个宏，实现 GPIO 的电平控制：

```
//CS 片选脚
#define MCP2510_CS_OUT
_raw_writel((_raw_readl(S3C2410_GPGCON)& (~(3<<4)))|(1<<4),S3C2410_GPGCON) /
#define MCP2510_CS_H
 raw writel( raw readl(S3C2410 GPGDAT)|(1<<2),S3C2410 GPGDAT)
#define MCP2510 CS L
_raw_writel(_raw_readl(S3C2410_GPGDAT)&(~(1<<2)),S3C2410_GPGDAT)//引脚
#define MCP2510 SI OUT
 raw writel(( raw readl(S3C2410 GPECON)&(~(3<<24)))|(1<<24),S3C2410 GPECON)
#define MCP2510_SI_H
_raw_writel(_raw_readl(S3C2410_GPEDAT)|(1<<12),S3C2410_GPEDAT)
#define MCP2510_SI_L
_raw_writel(_raw_readl(S3C2410_GPEDAT)&(~(1<<12)),S3C2410_GPEDAT)//SCK 引脚
#define MCP2510_SCK_OUT
_raw_writel((_raw_readl(S3C2410_GPECON)&(~(3<<26)))|(1<<26),S3C2410_GPECON)
#define MCP2510 SCK H
 raw writel( raw readl( S3C2410 GPEDAT)|(1<<13),S3C2410 GPEDAT)
#define MCP2510 SCK L
_raw_writel(_raw_readl(S3C2410_GPEDAT)&(~(1<<13)),S3C2410_GPEDAT)//SO 引脚
#define MCP2510 SO IN
_raw_writel((_raw_readl(S3C2410_GPECON)&(~(3<<22)))|(0<<22),S3C2410_GPECON)
#define MCP2510_SO_GET (_raw_readl(S3C2410_GPEDAT)&(1<<11))>>11
```

```
#define MCP2510 SO PULLUP
_raw_writel(_raw_readl(S3C2410_GPEUP)&(~(1<<11)),S3C2410_GPEUP)
#define MCP2510_SO_DISPULLUP
_raw_writel(_raw_readl(S3C2410_GPEUP)|(1<<11),S3C2410_GPEUP)
//中断
#define MCP2510_INT_IN
_raw_writel((_raw_readl(S3C2410_GPGCON)&(~(3<<0)))|(1<<1),S3C2410_GPGCON)
```

MCP2510_RW_Start 函数启动 SPI 读写：

```
void MCP2510 RW Start( void )
{
   MCP2510_SI_L ;
   MCP2510_SCK_L ;
   ndelay(400);
   MCP2510_CS_L ;
   ndelay(400);
}
```

SPI 接口写入数据完全是根据时序的要求，将每个字节按位送到总线上：

```
void Spi Write( unsigned char Data )
{
    unsigned char m ;
   //逐位写数据
    for( m = 0; m < 8; m++ )
    {
        if( (Data&0x80)==0x80 )
            MCP2510_SI_H;
        else
            MCP2510_SI_L;
        ndelay(400);
        MCP2510 SCK H ;
        Data = Data<<1 ;
        MCP2510 SCK L ;
        ndelay(400);
    }
}
```

SPI 接口数据读出函数实质就是按位读总线上的值：

```
unsigned char Spi_Read(void)
{
    unsigned char m ;
    unsigned char data = 0 ;
   //逐位读数据
    for( m = 0; m < 8; m++ )
```

```
    {
        MCP2510_SCK_H ;
        ndelay(400);
        data = data<<1;
        if( MCP2510_SO_GET != 0 )
            data |= 0x01 ;
        else
            data &= 0xfe;
        ndelay(400);
        MCP2510_SCK_L ;
        ndelay(400);
    }
    return (data);
}
```

向 MCP2510 指定地址写入一个字节包含两个步骤，先发送地址，再发送数据：

```
void MCP2510_Write( unsigned char address, unsigned char value)
{
    MCP2510_RW_Start() ;
    //实现 SPI 写时序
    Spi_Write(MCP2510INSTR_WRITE);
    Spi_Write( address );
    Spi_Write( value );
    MCP2510_CS_H ;
}
```

MCP2510_Read 实现从 MCP2510 指定地址中读出一个字节：

```
unsigned char MCP2510_Read( unsigned char address )
{
    unsigned char result;
    MCP2510_RW_Start() ;
    Spi_Write(MCP2510INSTR_READ);
    Spi_Write( address ) ;
    result = Spi_Read() ;
    MCP2510_CS_H ;
    return result ;
}
```

有了上面的基本函数，就可以控制 MCP2510 的所有寄存器了。所有的寄存器的访问都是对寄存器的相应的地址进行读写。通过中断来接收数据，首先要申请中断：

```
if (request_irq(IRQ_EINT8, &MCPCAN_interrupt, SA_INTERRUPT, "MCPCAN", NULL))
    {
            printk("request MCPCAN irq failed!\n");
            return -1;
    }
```

中断处理过程如下：

```
static irqreturn_t MCPCAN_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    unsigned char byte;
    //读中断源
    byte=MCP2510_Read(CANINTF);
#if MCP2510_DEBUG
    Uart_Printf("mcp2510 enter interrupt!: %d\n",byte);
#endif
    //中断来自第一个接收缓冲
    //if(byte & RX0INT){
    if( MCP2510_ReadStatus()&RX0INT )
    {
    MCP2510_Read_Can(3,&(mcp2510dev->MCP2510_Candata[mcp2510dev->receivePos]));
        MCP2510_WriteBits(CANINTF, ~RX0INT, RX0INT); //清除中断
#if MCP2510_DEBUG
        Uart_Printf("RX0INT!\n");
#endif
        if(MCPCAN_inc==1)//设备被应用程序打开，否则忽略
        {
         //指向下一个接收单元
            NextCanDataPos(mcp2510dev->receivePos);
          //唤醒等待
            wake_up_interruptible(&(mcp2510dev->wq));
        }
    }
    //中断来自第二个接收缓冲
    //if(byte & RX1INT){
    if( MCP2510_ReadStatus()&RX1INT )
    {
    MCP2510_Read_Can(4,&(mcp2510dev->MCP2510_Candata[mcp2510dev->receivePos]));
        MCP2510_WriteBits(CANINTF, ~RX1INT, RX1INT); //清除中断
#if MCP2510_DEBUG
        Uart_Printf("RX1INT!\n");
#endif
        if(MCPCAN_inc==1)
        {
            NextCanDataPos(mcp2510dev->receivePos);
            wake_up_interruptible(&(mcp2510dev->wq));
        }
    }
    return IRQ_HANDLED;
}
```

MCP2510 的报文是可以进行滤波接收的。MCP2510 可以设置多组滤波器，也就是说可以同时接收多种标示符不同的报文。下面介绍滤波实现，先定义一个结构：

```
struct MCP Filter
{
    unsigned int RXM0SIDH_ID;//屏蔽器 0 设置
    unsigned int RXM1SIDH_ID;//屏蔽器 1 设置
    unsigned int RXF0SIDH_ID;//滤波器 0 的 ID
    unsigned int RXF1SIDH_ID;//滤波器 1 的 ID
    unsigned int RXF2SIDH_ID;//滤波器 2 的 ID
    unsigned int RXF3SIDH_ID;//滤波器 3 的 ID
    unsigned int RXF4SIDH_ID;//滤波器 4 的 ID
    unsigned int RXF5SIDH_ID;//滤波器 5 的 ID
    //扩展标志
    unsigned char RXMSIDH_0;//屏蔽器 0 是否扩展帧
    unsigned char RXMSIDH_1;//屏蔽器 1 是否扩展帧
    unsigned char RXFSIDH_0;//滤波器 0 是否扩展帧
    unsigned char RXFSIDH_1;//滤波器 1 是否扩展帧
    unsigned char RXFSIDH_2;//滤波器 2 是否扩展帧
    unsigned char RXFSIDH_3;//滤波器 3 是否扩展帧
    unsigned char RXFSIDH_4;//滤波器 4 是否扩展帧
    unsigned char RXFSIDH_5;//滤波器 5 是否扩展帧
};
```

可以通过 IOCTL 实现滤波器设置：

```
int MCPCAN_ioctl(struct inode *inode, struct file *filp,unsigned int cmd, unsigned
long arg)
{
    unsigned char value=0;
    unsigned char ReadBackCNT = 0;
    void _user *argp = (void _user *)arg;
    switch(cmd)
    {
    case SET REVEIVEFILTER:
        if (copy_from_user(argp, (void _user *)arg, sizeof(struct MCP_Filter)))
        {
#if MCP2510 DEBUG
            printk("COMMAND_CHANGEFILTER ioctl error\n");
#endif
            return -EFAULT;
        }
        struct MCP_Filter*pfilter=(struct MCP_Filter*)argp;
        MCP2510_Write(MCP2510REG_CANCTRL, MODE_CONFIG);
        //最多等 8 次就退出
        while( ReadBackCNT<8 )
```

```
        {
            value = ( MCP2510_Read( MCP2510REG_CANSTAT ) & 0xe0 );
            if(value == MODE_CONFIG ){
#if MCP2510_DEBUG
                Uart_Printf( "ReadBackCNT = 0x%x\n", ReadBackCNT );
#endif
                break;
            }
            ReadBackCNT++ ;
        }
        //设置失败
        if( ReadBackCNT == 8 )
        {
            return -EFAULT;
        }
        //设置掩码
        MCP2510_Write_ID(RXM0SIDH, pfilter->RXM0SIDH_ID,pfilter->RXMSIDH_0);
        MCP2510_Write_ID(RXM1SIDH, pfilter->RXM1SIDH_ID,pfilter->RXMSIDH_1);
        //设置过滤的报文 ID
        MCP2510_Write_ID(RXF0SIDH, pfilter->RXF0SIDH_ID,pfilter->RXFSIDH_0);
        MCP2510_Write_ID(RXF1SIDH, pfilter->RXF1SIDH_ID,pfilter->RXFSIDH_1);
        MCP2510_Write_ID(RXF2SIDH, pfilter->RXF2SIDH_ID,pfilter->RXFSIDH_2);
        MCP2510_Write_ID(RXF3SIDH, pfilter->RXF3SIDH_ID,pfilter->RXFSIDH_3);
        MCP2510_Write_ID(RXF4SIDH, pfilter->RXF4SIDH_ID,pfilter->RXFSIDH_4);
        MCP2510_Write_ID(RXF5SIDH, pfilter->RXF5SIDH_ID,pfilter->RXFSIDH_5);
        //MCP2510_Write(CLKCTRL, MODE_LOOPBACK| CLKEN | CLK8);
        MCP2510_Write(CLKCTRL, MODE_NORMAL| CLKEN | CLK8);
        MCP2510_WriteBits(RXB0CTRL, (RXB_BUKT+RXB_RX_STDEXT), 0xFF);
        MCP2510_WriteBits(RXB1CTRL, RXB_RX_STDEXT, 0xFF);
#if MCP2510_DEBUG
        printk("MCPCAN ioctl successfully\n");
#endif
        break;
    case SET_CANBAUD:
        ...
        break;
    }
    return 0;
}
```

写数据的过程就是把数据放到发送缓冲寄存器中：

```
ssize_t MCPCAN_write(struct file *filp, const char _user *buf, size_t count,loff_
t *f_pos)
{
    char writeBuffer[sizeof(struct Canframe)];
```

```
    //数据长度是否满足要求
    if(count>=sizeof(struct Canframe)){
        //从用户空间复制数据
        copy_from_user(writeBuffer, buf, sizeof(struct Canframe));
        //将数据发送到 CAN 总线
        if(Mcp2510_Send_Data((struct Canframe*)writeBuffer)==0)
        {
            return sizeof(struct Canframe);
        }
    }
    return 0;
}
```

测试程序如下：

```
int fd;
struct Canframe scd_data;
struct Canframe rcd_data;
struct MCP Filter mcpfilter;
int retval;
int i=0;
//打开 CAN 设备
fd=open("/dev/can",O RDWR);
if(fd==-1)
{
    perror("error open\n");
    goto ERROR_EXIT;
}
printf("open /dev/can successfully\n");
//设置滤波器
mcpfilter.RXM0SIDH ID=0x1FFFFFFF;
mcpfilter.RXMSIDH 0=1;
mcpfilter.RXM1SIDH ID=0x1FFFFFFF;
mcpfilter.RXMSIDH 1=1;
mcpfilter.RXF0SIDH ID=10;
mcpfilter.RXFSIDH_0=1;
mcpfilter.RXF1SIDH_ID=0;
mcpfilter.RXFSIDH_1=0;
mcpfilter.RXF2SIDH_ID=11;
mcpfilter.RXFSIDH_2=1;
mcpfilter.RXF3SIDH_ID=0;
mcpfilter.RXFSIDH 3=0;
mcpfilter.RXF4SIDH ID=0;
mcpfilter.RXFSIDH 4=0;
mcpfilter.RXF5SIDH ID=0;
mcpfilter.RXFSIDH_5=0;
```

```
retval=ioctl(fd,SET REVEIVEFILTER,&mcpfilter);
if(retval<0)
{
    perror("ioctl error\n");
    goto ERROR_EXIT;
}
printf("ioctl successfully\n");
//下面的报文只可能在缓冲 0 里面收到
scd data.id=10;
scd data.dlc=3;
scd data.IsExt=1;
scd data.rxRTR=0;
memcpy(scd_data.data,"fgj",3);
//写数据
retval=write(fd,&scd_data,sizeof(struct Canframe));
if(retval==-1)
{
    perror("write error\n");
    goto ERROR EXIT;
}
else
{
    printf("write %d\n", retval);
}
//读数据
retval=read(fd,&rcd_data,sizeof(struct Canframe));
if(retval==-1)
{
    perror("read error\n");
    goto ERROR_EXIT;
}
else
{
    printf("read successfully:%s\n",rcd data.data);
}
 //下面的报文只可能在缓冲 1 里面收到
scd_data.id=11;
scd_data.dlc=5;
scd_data.IsExt=1;
scd_data.rxRTR=0;
memcpy(scd_data.data,"hello",5);
retval=write(fd,&scd data,sizeof(struct Canframe));
if(retval==-1)
{
    perror("write error\n");
```

```
        goto ERROR_EXIT;
    }
    else
    {
        printf("write %d\n", retval);
    }
    retval=read(fd,&rcd_data,sizeof(struct Canframe));
    if(retval==-1)
    {
        perror("read error\n");
        goto ERROR_EXIT;
    }
    else
    {
        printf("read successfully:%s\n",rcd_data.data);
    }
    ERROR_EXIT:
    close(fd);
    exit(-1);
```

测试结果如下：

#insmod can.ko
Using can.ko
MCP2510REG_CANCTRL = 0xE7
#mknod /dev/can c 223 0
#./demotest
open /dev/can sucReadBackCNT = 0x0
cessfully
MCPCAN ioctl successfully
ioctl successfully
mcp2510 enter interrupt!: 0
RX0INT!
write data size=24
id=a, ext=1,data=66,67,6a,0,4c,88,0,0
write 24re-1.read-0
read data size=24
id=a,ext=1, data=66,67,6a,0,0,0,0,0

read successfumcp2510 enter interrupt!: 255
RX1INT!
write data size=24
id=b, ext=1,data=68,65,6c,6c,6f,88,0,0
re-2.read-1
read data size=24
id=b,ext=1, data=68,65,6c,6c,6f,0,0,0

lly:fgj
write 24
read successfully:hello

4.4 Linux 的 I²C 驱动架构

Linux 对 I²C 设备提供了一个统一的驱动接口，在\drivers\i2c 目录下面。I²C 设备驱动用 i2c_driver 描述：

```
struct i2c_driver {
    struct module *owner;
    char name[32];
    int id;
    unsigned int class;
    unsigned int flags;
   //新总线出现
    int (*attach_adapter)(struct i2c_adapter *);
   //删除总线
    int (*detach_adapter)(struct i2c_adapter *);
   //删除一个从 I²C 设备
    int (*detach_client)(struct i2c_client *);
    //特殊指令处理，类似于 ioctl
    int (*command)(struct i2c_client *client,unsigned int cmd, void *arg);
   //设备驱动结构
    struct device driver driver;
    struct list head list;
};
int i2c_add_driver(struct i2c_driver *driver);
int i2c_del_driver(struct i2c_driver *driver);
```

I²C 适配器是用来直接与 I²C 设备打交道的，它包含了设备的访问算法、从设备的注册与注销等内容。

```
struct i2c_adapter {
    struct module *owner;
    unsigned int id;
    unsigned int class;//设备类型
    struct i2c_algorithm *algo;// 访问算法
    void *algo_data;
    //从设备注册与注销
    int (*client_register)(struct i2c_client *);
    int (*client_unregister)(struct i2c_client *);
    //同步控制
```

```
    struct semaphore bus lock;
    struct semaphore clist_lock;
    int timeout;//超时
    int retries;
    struct device dev;  //设备结构
    struct class_device class_dev; //设备类结构
    int nr;
    struct list_head clients;//从设备列表
    struct list_head list;
    char name[I2C_NAME_SIZE];
    struct completion dev_released;//设备释放完成结构
    struct completion class_dev_released;//设备类释放完成结构
};
int i2c_add_adapter(struct i2c_adapter *adap);
int i2c_del_adapter(struct i2c_adapter *adap);
```

I^2C 适配器的类型包括：

```
#define I2C_CLASS_HWMON         (1<<0)      // HWMON 传感器
#define I2C_CLASS_TV_ANALOG     (1<<1)      // 模拟电视卡
#define I2C_CLASS_TV_DIGITAL    (1<<2)      // DVB 电视卡
#define I2C_CLASS_DDC           (1<<3)      // i2c-matroxfb
#define I2C_CLASS_CAM_ANALOG    (1<<4)      // 模拟 CCD 摄像头
#define I2C_CLASS_CAM_DIGITAL   (1<<5)      // 网络摄像机
#define I2C_CLASS_SOUND         (1<<6)      // 声音设备
#define I2C_CLASS_ALL           (UINT_MAX)  // 上述所有
```

I^2C 访问算法的结构：

```
struct i2c algorithm {
    char name[32];//算法名
    unsigned int id;
    //I2C 主设备数据收发
int (*master_xfer)(struct i2c_adapter *adap,struct i2c_msg msgs[], int num);
    //SMBus 主设备数据收发
int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr, unsigned short flags, char
read_write,u8 command, int size, union i2c_smbus_data * data);
    //从设备收发接口
    int (*slave_send)(struct i2c_adapter *,char*,int);
    int (*slave_recv)(struct i2c_adapter *,char*,int);
    //特殊指令处理，类似于 ioctl
    int (*algo_control)(struct i2c_adapter *, unsigned int, unsigned long);
    //决定本适配器支持的功能
    u32 (*functionality) (struct i2c_adapter *);
};
```

I^2C 客户结构代表一个连接到 I^2C 总线上的设备（如芯片）。

```
struct i2c_client {
```

```
    int id;
    unsigned int flags;                     //I²C 客户设置
#define I2C_CLIENT_ALLOW_USE 0x01           //客户允许访问
#define I2C_CLIENT_ALLOW_MULTIPLE_USE  0x02//允许对一个 I²C 客户的多个带锁访问
#define I2C_CLIENT_PEC  0x04                //使用包错误检查
#define I2C_CLIENT_TEN  0x10                //10 位地址
    unsigned int addr;                      //芯片地址，注意是低 7 位
    struct i2c_adapter *adapter;            //关联的 adapter
    struct i2c_driver *driver;              //对应的 i2c_driver
    int usage_count;                        //访问个数
    struct device dev;                      //设备结构
    struct list_head list;
    char name[I2C_NAME_SIZE];
    struct completion released;
};
```

Linux 中的 I^2C 驱动包括 I^2C 核心层、I^2C 总线驱动和 I^2C 设备驱动，I^2C 总线驱动只是提供了对一条总线的读写机制，实际的 I^2C 通信是由 I^2C 设备驱动来完成的，Linux 下 I^2C 驱动结构如表 4.4 所示。

表 4.4 Linux 下 I^2C 驱动结构

层次	作用	代码
核心管理层 I^2C core	实现对 I^2C 总线、I^2C adapter 及 I^2C driver 的管理	i2c-core.c
I^2C 总线驱动	实现 I^2C adapter、i2c_algorithm	\drivers\i2c\algos \drivers\i2c\busses
I^2C 设备驱动	针对特定的 I^2C 设备，实现 i2c_driver 层的功能和对 i2c_client 的管理	\drivers\i2c\chips \drivers\i2c\i2c-dev.c

\drivers\i2c\i2c-dev.c 是一个典型的 I^2C 设备驱动，它提供了统一的应用层访问接口，即 read，write 以及 ioctl 等文件操作。所有的 I^2C 设备都通过 i2C-dev 中定义的文件接口访问。

```
static struct file_operations i2cdev_fops = {
    .owner      = THIS_MODULE,
    .llseek     = no_llseek,
    .read       = i2cdev_read,
    .write      = i2cdev_write,
    .ioctl      = i2cdev_ioctl,
    .open       = i2cdev_open,
    .release    = i2cdev_release,
};
```

内核在 i2c_dev_init(void)中将 I^2C 的驱动与 i2cdev_fops 关联起来了。如果要开发自己的 I^2C 驱动，可以参考 i2c-dev.c。

```
res = register_chrdev(I2C_MAJOR, "i2c", &i2cdev_fops);
```

i2cdev_write 的处理过程为：

```
static ssize_t i2cdev_write (struct file *file, const char _user *buf, size_t
count,loff_t *offset)
```

```
{
    int ret;
    char *tmp;
    struct i2c_client *client = (struct i2c_client *)file->private_data;
    //最大只处理 8KB 数据
    if (count > 8192)
        count = 8192;
    tmp = kmalloc(count,GFP KERNEL);
    if (tmp==NULL)
        return -ENOMEM;
    if (copy_from_user(tmp,buf,count)) {
        kfree(tmp);
        return -EFAULT;
    }
    pr_debug("i2c-dev: i2c-%d writing %zd bytes.\n",
        iminor(file->f dentry->d inode), count);
    ret = i2c_master_send(client,tmp,count);
    kfree(tmp);
    return ret;
}
```

显然 i2cdev_write 先将用户数据复制到内核空间后，再调用 i2c_master_send：

```
Int i2c_master_send(struct i2c_client *client,const char *buf ,int count)
{
    int ret;
    struct i2c_adapter *adap=client->adapter;
    struct i2c msg msg;
    //判断函数是否定义
    if (client->adapter->algo->master_xfer) {
        msg.addr  = client->addr;
        msg.flags = client->flags & I2C_M_TEN;
        msg.len = count;
        msg.buf = (char *)buf;
        dev dbg(&client->adapter->dev, "master send: writing %d bytes.\n",
            count);
        down(&adap->bus lock);
        ret = adap->algo->master_xfer(adap,&msg,1);
        up(&adap->bus_lock);
        //返回结果
        return (ret == 1 )? count : ret;
    } else {
        dev err(&client->adapter->dev, "I2C level transfers not supported\n");
        return -ENOSYS;
    }
}
```

最后是调用算法中的 master_xfer 函数来实现。可以看到 master_xfer 的第二个参数是 struct

i2c_msg，相信熟悉 I^2C 时序的人都知道它的意义。第三个参数就是消息的个数。master_xfer 中实际就是实现了 I^2C 发送数据时序。

```
struct i2c_msg {
    _u16 addr;/*从设备地址*/
    _u16 flags;//芯片地址长度
    _u16 len;/* 消息长度*/
    _u8 *buf;/* 消息数据*/
};
```

注意上面结构中的 flags，当设置 I2C_M_RD 位时为读，否则为写，所以在内核中 master_xfer 函数既实现了 I^2C 读，还实现了 I^2C 写。它的各位有如下意义：

```
#define I2C_M_TEN            0x10//采用 10 比特的地址位
#define I2C_M_RD             0x01//读标志
#define I2C_M_NOSTART        0x4000 //舍弃起始位
#define I2C_M_REV_DIR_ADDR   0x2000//对读写标志进行互换
#define I2C_M_IGNORE_NAK     0x1000//将无应答作为应答处理
#define I2C_M_NO_RD_ACK      0x0800//舍弃读应答
```

4.5　Linux I^2C 驱动开发

Linux 下的 I^2C 驱动开发包括两种类型，一类是总线类驱动，一类是设备类驱动。对于设备层，一般来说，可以参照 i2c-dev.c。i2c-dev.c 已经实现了 I^2C 设备的文件操作接口（struct file_operations i2cdev_fops），只要实现 struct i2c_drive 就可以了。例如 ds1621 的驱动（\drivers\i2c\chips\ds1621.c）：

```
static struct i2c_driver ds1621_driver = {
    .owner          = THIS_MODULE,
    .name           = "ds1621",
    .id             = I2C_DRIVERID_DS1621,
    .flags          = I2C_DF_NOTIFY,
    .attach_adapter = ds1621_attach_adapter,
    .detach_client  = ds1621_detach_client,
};
i2c_add_driver(&ds1621_driver);
```

关键是实现 ds1621_attach_adapter 和 ds1621_detach_client 等函数了，这里不一一分析了，有兴趣的读者可以阅读内核代码。

对于特定的硬件平台，需要增加相应的 I^2C 总线驱动来支持 I^2C 设备。设备层驱动通过 i2c_driver 的 id 来区分。下面以 PCF8584 为例说明如何开发 I^2C 总线驱动。一个 I^2C 总线驱动通常需要两个模块来描述，即一个 struct i2c_adapter（i2c-elektor.c）和一个 struct i2c_algorithm（i2c-algo-pcf.c）：

```
static struct i2c_algo_pcf_data pcf_isa_data = {
    .setpcf      = pcf_isa_setbyte,
```

```
    .getpcf      = pcf_isa_getbyte,
    .getown      = pcf_isa_getown,
    .getclock    = pcf_isa_getclock,
    .waitforpin = pcf_isa_waitforpin,
    .udelay      = 10,
    .mdelay      = 10,
    .timeout     = 100,
};//操作函数结构
static struct i2c_algorithm pcf_algo = {
    .name            = "PCF8584 algorithm",
    .id              = I2C_ALGO_PCF,
    .master_xfer     = pcf_xfer,
    .functionality  = pcf_func,
};//算法接口
static struct i2c_adapter pcf_isa_ops = {
    .owner           = THIS_MODULE,
    .id              = I2C_HW_P_ELEK,
    .algo_data       = &pcf_isa_data,
    .name            = "PCF8584 ISA adapter",
};//适配器
```

在 I²C-Bus adapter routines for PCF8584 ISA bus adapter 模块的 i2c_pcfisa_init 中调用 i2c driver algorithms for PCF8584 adapters 中的 i2c_pcf_add_bus 函数注册 pcf_isa_ops。

```
static int _init i2c_pcfisa_init(void)
{
    ...
    if (i2c_pcf_add_bus(&pcf_isa_ops) < 0)goto fail;
}
```

i2c_pcf_add_bus 的代码如下：

```
int i2c_pcf_add_bus(struct i2c_adapter *adap)
{
    struct i2c_algo_pcf_data *pcf_adap = adap->algo_data;
    int rval;
    DEB2(dev_dbg(&adap->dev, "hw routines registered.\n"));
    //向 I²C 模块注册新的适配器
    adap->id |= pcf_algo.id;
    adap->algo = &pcf_algo;
    //设置超时值和重试次数
    adap->timeout = 100;
    adap->retries = 3;
    //初始化
    rval = pcf_init_8584(pcf_adap);
    if (!rval)
        i2c_add_adapter(adap);
    return rval;
}
```

第5章 USB驱动程序

Linux 内核支持几乎所有的通用 USB 设备，包括键盘、鼠标、打印机、MODEM、摄像头、游戏杆、电视盒、扫描仪、网卡等。在主机控制器方面，Linux 内核支持 USB 1.1 的 UHCI 与 OHCI 和 USB 2.0 的 EHCI。另外，Linux 内核还提供了作为从设备的 USB Gadget 驱动。本章将介绍 USB 基础知识、USB 驱动框架、USB 摄像头驱动、USB Gadget 驱动等方面的相关内容。

5.1 USB 总线

5.1.1 USB 总线概述

通用串行总线 USB 具有热插拔、即插即用、数据传输可靠、扩展方便、低成本等优点，广泛用于计算机接口和各种嵌入式系统。USB 1.X 支持两种总线速率：全速（12Mbps）和低速（1.5Mbps）。定义低速模式是为了支持少量的低带宽设备，如鼠标、键盘等。2000 年 4 月 USB 2.0 版本被推出，它的最高速率可达 480Mbps，是 USB 1.1 协议的 40 倍，这个飞跃使该接口可以面向更多的应用。USB 2.0 支持三种数据传送速率：（1）USB 高速，480Mbps；（2）USB 全速，12Mbps；（3）USB 低速，1.5Mbps。当然，总线的速率并不等于设备真正传送数据的速率。数据传送速率要依靠总线的繁忙程度，以及它所使用的数据传输类型。

USB 的物理接口包括电气和机械两方面规范。电气方面，USB 是通过一条含 4 根导线（一对信号线和一对电源线）的电线来传输信号和电源的，如图 5.1 所示。在 USB 设备与主机之间通过两根导线（D+和 D–）传送信号。在主机控制器和集线器之间可以高速传送全速和低速设备的数据，而在集线器和设备之间全速和低速传送数据。这种性能减少了全速或低速设备对高速设备带宽的影响。USB 采用位填充 NRZI 码方案，每个数据包之前是 SNYC 域，用于同步位时钟。VBUS 和 GND 线用于向设备传送电源。VBUS 通常是+5V 电压，USB 总线的电缆如图 5.1 所示。

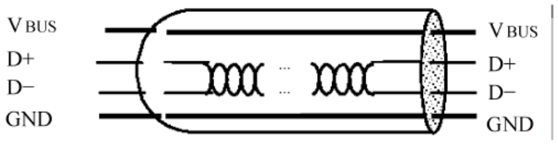


图 5.1　USB 总线的电缆

5.1.2 USB 系统组成

一个 USB 系统一般由一个 USB 主机、一个或多个 USB 集线器和一个或多个 USB 设备节点组成。在 USB 总线上，USB 会为每个连接在总线上的 USB 设备分配一个地址，USB 主机通过设

备地址访问相关的设备。USB 总线连接了 USB 设备和 USB 主机，USB 的物理连接是有层次性的星型结构。集线器能够增加外设的端口数。一个典型的 USB 集线器包括一个集线器中继器、一个集线器控制器和多个上下游端口，具有 USB 设备连接、电源管理和总线错误检测等功能。一个 USB 2.0 集线器则相当于一个远程处理器，可以根据需要实现从高速到低速或全速的转换，USB 系统拓扑结构如图 5.2 所示。

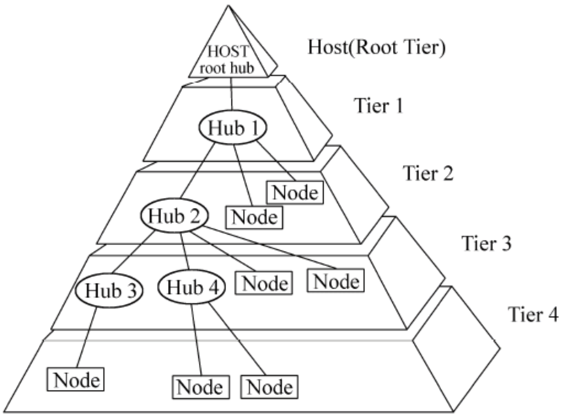


图 5.2　USB 系统拓扑结构

1．USB 主机（Host）

在任何 USB 系统中仅有一台主机（Host）。主机系统中的 USB 接口称为主机控制器（Host Controller）。主机控制器可以由硬件、固件或软件结合实现。根集线器（Hub）集成在主机系统中，以提供一个和多个连接点。USB 主机通过主机控制器与 USB 设备进行交互。USB 主机负责的任务包括：（1）检测 USB 设备的连接和拆除；（2）管理主机和 USB 设备之间的控制流；（3）管理主机和 USB 设备之间的数据流；（4）收集状态和活动的统计；（5）为连接的 USB 设备提供电源。主机上的 USB 系统软件用于管理 USB 设备和基于主机的设备软件之间的交互操作。USB 系统软件和设备软件之间的交互操作包括设备的枚举和配置、同步数据传送、异步数据传送、电源管理以及设备和总线的管理信息等 5 个方面。

2．USB 设备（Node）

所有的 USB 设备都是通过 USB 地址来存取的，这个地址在连接或枚举时分配。USB 设备对于 USB 系统来说是一个端点的集合，端点被分成组，一组端点实现一个接口，设备端点和主机软件之间利用管道进行联系。设备驱动程序就是通过这些接口和管道来与设备进行通信的。

设备端点是一个 USB 设备中唯一可寻址的部分，是主机与设备之间通信的来源或目的。它是主机与设备间通信流的一个结束点。一系列相互独立的端点在一起构成了 USB 逻辑设备。每个逻辑设备有一个唯一的地址，这个地址是在设备连上主机时由主机分配的，而设备中的每个端点在设备内部有唯一的端点号。这个端点号是在设备设计时被给定的。每个端点都是一个简单的连接点，或者支持数据流进设备，或者支持其流出设备，两者不可兼得。一个端点的特性决定了它与客户软件进行的传送的类型。端点号不为 0 的端点在被设置前处于未知状态，是不能被主机访问的。缺省控制通道支持了对控制的传送，一旦设备接上，并加电，且又收到一个总线复位命令，端点 0 就是可访问的了。设备可以有除 0 以外的其他端点，这取决于这些设备的实现。低速设备在 0 号输入及输出端点外，只能有两个额外的可选端点。而高速设备可具有的额外端点数仅受限于协议的定义（协议中规定，最多 15 个额外的输入端点和最多 15 个额外的输出端点）。除缺省控制通道的默认端点外，其他端点只有在设备被设置后才可使用，对设备的设置是设备设置过程的一部分。

主机与设备上的端点之间的 USB 数据传送模式称为管道（Pipe）。有两种类型的管道：流（Stream）和消息（Message）。流数据是无结构的，而消息数据是有结构的。另外，管道是数据带

宽传输服务类型和端点特性（如方向性和缓冲器大小）的结合。大部分管道在 USB 设备配置时即存在。一个消息管道（即默认的控制管道）通常在一个设备供电时即存在，这是为了可以存取设备配置状态和控制信息。事务调度允许为某些流管道进行流量控制。在硬件方面，通过 NAK 及握手控制数据传输速度，防止缓冲区出现过速和过缓的情况。

由两个 0 号端点组成的通道叫缺省控制通道。一旦设备加电并复位后，此通道即可使用。其他通道只在设备被设置后才存在。USB 系统软件在决定设备身份、设置要求和设置设备时使用缺省控制通道。当设备被设置后，这个设备的特定软件还可使用该通道。USB 系统软件保留缺省控制通道的拥有权，协调其他客户软件对通道的使用。

3. USB 集线器（Hub）

USB HUB 用于设备扩展连接，所有 USB DEVICE 都连接在 USB HUB 的端口上。一个 USB HOST 总与一个根 HUB（USB ROOT HUB）相连。

5.1.3 USB 传输模式

主控制器负责主机和 USB 设备间数据流的传输。这些传输数据被当作连续的比特流。每个设备提供了一个或多个可以和客户程序通信的接口。每个接口由 0 个或多个管道组成，这些管道分别独立地在客户程序和设备的特定终端间传输数据。USB 支持 4 种基本的数据传输模式：控制传输、同步传输、中断传输、批量传输。每种传输模式使用到具有相同名字的端点，具有不同的传输特性。USB 总线属于一种轮询方式的总线，由主端口预定的标准协议使各从设备分享 USB 带宽。在每次传送开始时，主控制器发送一个描述传输运作的种类、方向、USB 设备地址和终端号的 USB 数据包，这个数据包通常称为标志包（token packet）。USB 设备从解码后数据包的适当位置取出属于自己的数据。

1. 控制传输方式（Control）

控制传输方式支持双向传输；用来处理主端口到 USB 从端口的数据传输，包括设备控制指令、设备状态查询及确认命令。当 USB 设备收到这些数据和命令后，将依据先进先出的原则处理到达的数据。其传输的最大负荷与中断传输方式相同。对于高速设备，允许数据包最大容量为 8，16，32 或 64 字节，对于低速设备只有 8 字节一种选择。

2. 同步传输方式（Synchronous）

同步传输是一种周期的、连续的单向传输方式，通常用于与时间有密切关系的信息的传输。该方式占用预先分配好的带宽，并且有预定发送延时，用来连接需要连续传输数据且对数据的正确性要求不高而对时间极为敏感的外部设备。在传送数据发生错误时，USB 并不处理这些错误，而是续传新的数据。同步传输每次传输的最大有效负荷可为 1024 字节。

3. 中断传输方式（Interrupt）

中断传输用于非周期的、自然发生的、数据量很小的信息的传输。数据传送方向是从设备到主机。此方式主要用在键盘、鼠标及操纵杆等设备上。全速设备每次中断传输的最大有效负荷可

为 64 个字节，而低速设备每次中断传输的最大有效负荷仅为 8 个字节。

4．批量传输方式（Bulk）

批量传输方式也是一种单向传输，用于大量的、对时间没有要求的数据传输。如果一个外设需要双向传输，则必须使用另一个端点；该方式用来传输要求正确无误的数据。通常打印机、扫描仪和数码相机以这种方式与主机连接。在数据相对比较多和突发数据量较大时使用，在传输限制方面具有很宽的动态自由度。批量传输每次数据传输的最大有效负荷可为 64 个字节。

5.1.4　主机规范

USB 设备作为一个完整的硬件设备，是由硬件和固件两部分组成的。其中固件中包括了有关系统配置和 CPU 的一些设置模块、USB 协议栈模块等几部分。USB 总线上的信息有两种：一种是差模数据线上的包；另一种则是有特殊定义的数据线的信号，比如复位信号、远程唤醒信号等。因此，设备的 USB 栈就要能够识别并处理这些不同的信息内容。同时，在上层，这些信息又要被组成各种传输的类型来加以处理。所以，整个协议栈的内容是非常庞大的。

USB 主机是 USB 总线的核心部分。它负责管理整个 USB 总线的所有信息。为了更好地实现 USB 主机的功能，USB 厂商提出了集中不同的主机控制器的设计规范，USB 1.1 中包括 OHCI（Open Host Controller Interface）和 UHCI（Universal Host Controller Interface）。USB 2.0 中为 EHCI（Enhanced Host Controller Interface）。

EHCI 从寄存器级对 USB 2.0 主机高速数据传输控制器进行了详细描述。它为 USB 2.0 主机高速数据传输控制器的软硬件设计提供了统一的接口标准，这大大简化了 USB 2.0 的主机设计，提高了软件的可移植性。为了兼容 USB 1.1，USB 2.0 的 HC 由 EHCI 和 CHC（Companion Host Controller 包括 OHCI 和 UHCI 等）两部分组成。EHCI 包含三个接口空间：PCI 配置空间、HC 寄存器空间和调度接口空间。PCI 配置空间主要包含 HC 的 PCI 接口相关的配置管理；HC 寄存器空间主要由 EHCI 控制寄存器和状态寄存器组成，该空间一般可以作为 I/O 空间直接访问或通过内存映射机制映射成可直接操作的 I/O 空间。调度接口空间主要提供异步数据传输和周期性数据传输。

5.1.5　USB 设备描述符

USB 设备在逻辑上分成了几个层次，分别是设备层、配置层、接口层和端点层。USB 设备中各层关系如图 5.3 所示。主机识别一个 USB 设备必须经过枚举的过程，主机使用总线枚举来识别和管理必要的设备状态变化。总线枚举的过程如下所示。

（1）设备连接：USB 设备接入 USB。

（2）设备上电：USB 设备可以使用 USB 总线供电，也可以使用外部电源供电。

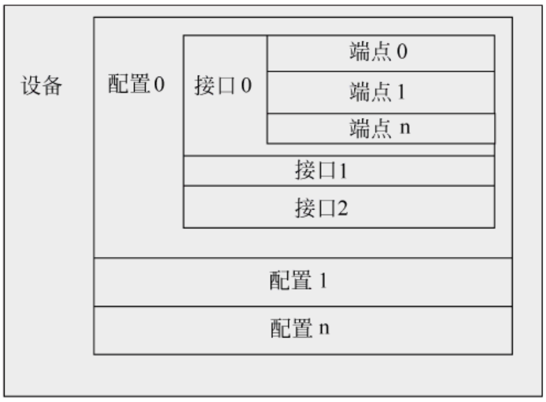


图 5.3　USB 设备中各层关系

（3）主机检测到设备，发出复位：设备上电后，主机通过设备的上拉电阻检测到有新的设备连接，主机向该端口发送一个复位信号。

（4）设备默认状态：设备要从总线上接收到一个复位的信号后，才可以对总线的处理操作做出响应。设备接收到复位信号后，就使用默认地址（00H）来对其进行寻址。

（5）地址分配：当主机接收到有设备对默认地址（00H）的响应的时候，就对设备分配一个空闲的地址，设备以后就只对该地址进行响应。

（6）读取 USB 设备描述符：主机读取设备描述符，确认 USB 设备的属性。

（7）设备配置：主机依照读取的 USB 设备描述符来进行配置，如果设备所需的 USB 资源得以满足，就发送配置命令给 USB 设备，表示配置完毕。

（8）挂起：为了节省电源，当总线保持空闲状态超过 3ms 以后，设备驱动程序就会进入挂起状态。挂起状态时设备的消耗电流不超过 500μA。当被挂起时，USB 设备保留了包括其地址和配置信息在内的所有内部状态。

完成以上的几步工作后，USB 设备就可以使用了。在枚举的过程中，USB 设备通过设备描述符说明其相关属性和与 USB 主机之间数据传输的方式。USB 设备都必须支持 USB 规范定义的标准命令，USB 主机必须提供对 USB 设备的配置和管理工作。

1. 设备描述符

设备描述符给出了 USB 设备的一般信息。这包括对设备及所有设备配置起全程作用的信息，如图 5.4 所示。一个 USB 设备只能有一个设备描述符。所有的 USB 设备都有默认控制通道。默认控制通道的最大包长在设备描述符中得到了说明。

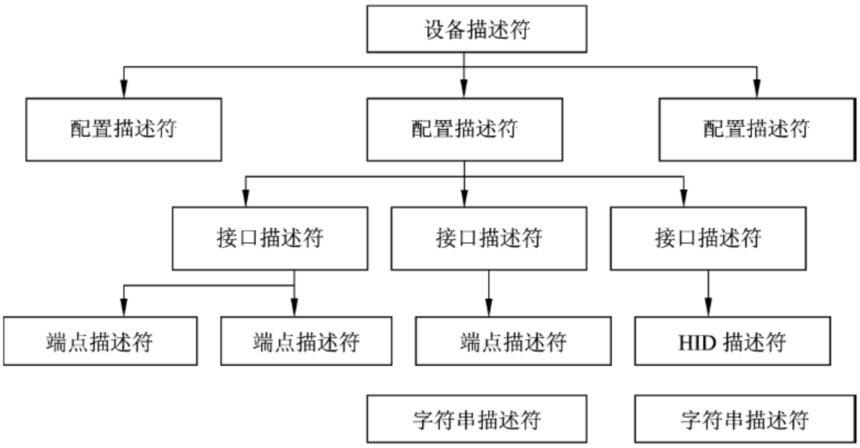


图 5.4　USB 设备的描述符

```
struct usb_device_descriptor {
    _u8  bLength; //此描述符的字节数
    _u8  bDescriptorType;// 描述符种类为设备
    _u16 bcdUSB;// 此设备与描述符兼容的 USB 设备说明版本号（BCD 码）
    _u8  bDeviceClass;// 设备类码
    _u8  bDeviceSubClass;// 设备子类码
    _u8  bDeviceProtocol;// 协议码
```

```
        _u8  bMaxPacketSize0;// 端点 0 的最大包大小（仅 8,16,32,64 为合法值）
        _u16 idVendor;// 厂商标志
        _u16 idProduct;// 产品标志
        _u16 bcdDevice;// 设备发行号（BCD 码）
        _u8  iManufacturer;// 描述厂商信息的字串的索引
        _u8  iProduct;// 描述产品信息的字串的索引
        _u8  iSerialNumber;// 描述设备序列号信息的字串的索引
        _u8  bNumConfigurations;//此设备支持的配置数
} _attribute_((packed));
```

设备类码的典型值如下：

```
#define USB_CLASS_PER_INTERFACE     0
#define USB_CLASS_AUDIO             1//声音设备
#define USB_CLASS_COMM              2//调制解调设备、网卡、ISDN 连接
#define USB_CLASS_HID               3 //HID 设备，如鼠标、键盘
#define USB_CLASS_PHYSICAL          5//物理设备
#define USB_CLASS_STILL_IMAGE       6//静止图像捕捉设备
#define USB_CLASS_PRINTER           7//打印机
#define USB_CLASS_MASS_STORAGE      8//批量存储设备
#define USB CLASS HUB               9// USB Hubs
#define USB CLASS CDC DATA          0x0A
#define USB_CLASS_CSCID             0x0B//智能卡
#define USB_CLASS_CONTENT_SEC       0x0D//内容安全
#define USB_CLASS_VIDEO             0x0E//视频设备，例如网络摄像头
#define USB_CLASS_APP_SPEC          0xFE//应用自定义设备
#define USB_CLASS_VENDOR_SPEC       0xFF//厂商自定义设备
```

2. 配置描述符

配置描述符给出了 USB 设备的配置信息，配置描述符包括一个 bConfigurationValue 域，在 SetConfiguration()请求时被用作参数来设置所需配置。此描述符给出了此配置下的接口数、每个接口可能的独立操作。一个 USB 设备有一个或多个配置。每个配置有一个或多个接口。而每个接口又有 0 个或多个端节点。在一个配置下，一个端节点不会在接口之间共享，除非端节点被同一个接口的不同配置使用。当主机发出请求，要求获取配置描述符时，所有相关接口与端节点的描述符都被返回。

```
struct usb config descriptor {
        _u8  bLength;// 此描述符的字节数
         u8  bDescriptorType;// 配置描述符类型
        _u16 wTotalLength;// 此配置信息的总长（包括配置，接口，端点和设备类及厂商定义的描述符）
        _u8  bNumInterfaces;// 此配置所支持的接口个数
        _u8  bConfigurationValue;// 在 SetConfiguration()请求中用作参数来选定此配置
        _u8  iConfiguration;// 描述此配置的字串描述符索引
        _u8  bmAttributes;// 电源配置特性，一个既用总线电源又有自给电源的设备会在 MaxPower 域
```

```
指出需要从总线取的电量。并设置 D6 为 1。运行时期的实际电源模式可由 GetStatus(DEVICE) 请求得到
        __u8  bMaxPower;// 在此配置下的总线电源耗费量。以 2mA 为一个单位。
    } __attribute__ ((packed));
```

3. 接口描述符

接口描述符在一个配置内给出一个接口的信息。如果一个配置支持不只一个接口，端节点的描述符会跟在接口描述符后被返回，接口描述符总是作为配置描述符的一部分被返回。接口描述不可直接用 SetDescription ()和 GetDescriptor ()存取。

一个接口可能包含备选设置，以使得端节点或它们的特性在设备配置好以后能改变。一个接口的默认设置总是可选设置。SetInterface ()与 GetInterface ()用来选择与返回选择了的接口设置。

可选的接口设置使得部分的设备配置能在其他接口进行操作的情况下改变。如果一个配置对于它的一个或多个接口有备选设置，每一设置包括一个独立接口描述符和相关节点。

如果一个设备配置支持单个接口，并且此接口有两个可选设置，配置描述符返回以后会紧跟着返回 bInterfaceNumber 与 bAlternateSetting 域皆为 0 的第一个设置的接口描述符及相关的节点描述符，而随之后是另一个设置接口描述符与节点描述符。第二个接口描述符的 bInterfaceNumber 域也应为 0，但 bAlternateSetting 域应为 1。

如果一个接口仅使用节点 0，则接口描述符以后就不再返回节点描述符，并且此接口表示的是一个请求接口，它使用连在节点 0 上的默认通道。在这种情况下 bNumberEndpoints 域应被设置成 0。一个接口描述符的节点个数不把节点 0 计在内。

```
    struct usb_interface_descriptor {
        __u8  bLength;// 此描述符的字节数
        __u8  bDescriptorType; //接口描述符类
        __u8  bInterfaceNumber;// 接口号: 当前配置支持的接口数组索引，从零开始
        __u8  bAlternateSetting;// 可选设置的索引值
        __u8  bNumEndpoints;// 此接口用的端点数量，如果是 0 则说明此接口只用缺省控制管道
        __u8  bInterfaceClass;//类值: 0 值为将来的标准保留。如果此域的值设为 0FFH，则此接口类
                                //由厂商说明。所有其他的值由 USB 说明保留
        __u8  bInterfaceSubClass;// 子类码
        __u8  bInterfaceProtocol;// 协议码
        __u8  iInterface;// 描述此接口的字串描述符的索引值
    } __attribute__ ((packed));
```

4. 端口描述符

每个接口使用的端口都有自己的描述符，此描述符被主机用来决定每个端口的带宽需求。每个端口的描述符总是作为配置描述的一部分返回的。

```
    struct usb_endpoint_descriptor {
        __u8  bLength;// 此描述符的字节数
        __u8  bDescriptorType;// 端点描述符类
```

```
      _u8  bEndpointAddress;// 此描述符所描述的端点的地址
      _u8  bmAttributes;// 此域的值描述的是在 bConfigurationValue 域所指的配置下端点的特
      //性。Bit[1,0]代表传送类型: 00=控制传送; 01=同步传送; 10=批传送; 11=中断传送。所有其
      //他的位都保留
      _u16 wMaxPacketSize;// 当前配置下此端点能够接收或发送的最大数据包的大小
      _u8  bInterval;// 轮寻数据传送端点的时间间隙
      //以下用于声音端点
       u8  bRefresh;
       u8  bSynchAddress;
} _attribute_ ((packed));
```

5. 字串描述符

字串描述符是可有可无的。如前所述，如果一个设备无字串描述符，所有其他描述符中有关字串描述符的索引都必须为 0。

```
struct usb string descriptor {
      _u8  bLength; // 此描述符的字节数
      _u8  bDescriptorType;
      _u16 wData[1];//    UTF-16LE 编码数据
} _attribute_ ((packed));
```

USB 2.0 设备增加了设备限定描述符和其他速度配置描述符，这两个新添加的描述符可以通过标准的 GETDESCRIPTOR 请求来读取。USB 2.0 的端点描述符与 USB 1.x 的规范格式相同，只是添加了高速传输的属性。

6. 设备限定描述符

设备限定描述符的相关内容见表 5.1。

表 5.1　设备限定描述符

偏移量	域	大小	值	描述
0	bLength	1	数字	描述符的字节数长度
1	bDescriptorType	1	常量	设备限定类型（0x06）
2	bcdUSB	2	BCD 码	USB 设备版本号 0x0200
4	bDeviceClass	1	类	类代码
5	bDeviceSubClass	1	子类	子类代码
6	bDeviceProtocol	1	协议	协议代码
7	bMaxPacketSize0	1	数字	其他速度的最大数据包大小
8	bNumConfigurations	1	数字	其他速度相关配置描述符数目
9	bReserved	1	0	保留，为 0

7. 其他速度配置描述符

其他速度配置描述符的相关内容见表 5.2。

表 5.2　其他速度配置描述符

偏移量	域	大小	值	描述
0	bLength	1	数字	描述符的字节数长度
1	bDescriptorType	1	常量	其他速度配置描述符类型（0x07）
2	wTotalLength	2	数字	返回数据的总长度
4	bNumInterface	1	数字	该配置支持的接口数量
5	bConfigurationValue	1	数字	选择当前配置描述符
6	iConfiguration	1	索引	字符串描述符索引
7	bmAttributes	1	位图	电源特性描述
8	bMaxPower	1	毫安	最大耗电量

USB 协议规定所有的设备必须响应如表 5.3 所示的标准请求。USB 设备在设备的默认控制通道（Default Control Pipe）处对主机的请求发出响应。这些请求是通过使用控制传输来达到的。

表 5.3　USB 标准设备请求

请求类型	设备请求	值	索引	长度	数据
10000000B 10000001B 10000010B	GetStatus(00H)	0	设备 接口 端点	2	设备、接口或端点的状态
00000000B 00000001B 00000010B	ClearFeature(01H)	特性选择符	设备 接口 端点	0	无
10000000B 10000001B 10000010B	SetFeature (03H)	特性选择符	设备 接口 端点	0	设备、接口或端点的状态
00000000B	SetAddress(05H)	设备地址	0	0	无
10000000B	GetDescriptor(06H)	描述符类型	0 或语言 ID	描述符长度	描述符
00000000B	SetDescriptor(07H)	描述符类型	0 或语言 ID	描述符长度	描述符
10000000B	GetConfiguration(08H)	0	0	1	配置值
00000000B	SetConfiguration(09H)	配置值	0	0	无
10000000B	GetInterface(0AH)	0	接口	1	可选接口
00000000B	SetInterface(0BH)	可选配置	接口	0	无
10000010B	SynchFrame(0CH)	0	端点	2	帧标号

5.1.6　HID 类规范

HID 是 Human Interface Device 的缩写，是直接与人交互的设备，如键盘、鼠标与游戏杆等。但是 HID 设备并不一定要有人机接口，只要符合 HID 类别规范的设备都是 HID 设备。所有的 HID 类设备通过 USB 默认的控制管道和中断管道与 USB 主机通信。控制管道用于传输 USB 描述符、类请求代码以及供查询的消息数据等。中断管道主要用于传输数据。HID 类的描述符由 5 个标准描述符（设备描述符、配置描述符、接口描述符、端点描述符、字符串描述符）和三个 HID 类描

述符（HID 描述符、报表描述符、实体描述符）组成。USB HID 设备请求包括标准的 USB 设备请求和 HID 类请求。HID 类请求包括 Get_Report，Get_Idle，Get_Protocol，Set_report，Set_Idle，Set_Protocol。HID 类设备通过报表描述符与主机进行数据传输。表 5.4 是 HID 描述符。

表 5.4　HID 描述符

偏移量	域	大小	值	描述
0	bLength	1	数字	描述符的字节数长度
1	bDescriptorType	1	常量	描述符类型（0x21）
2	bcdHID	2	数字	HID 规范版本
4	bcountryCode	1	数字	国家识别号码
5	bNumDescriptors	1	数字	支持的附属描述符数目
6	bDescriptType	1	常量	类别描述符的类型
7	wDescriptorLength	2	数字	报表描述符总长度
9	[bDescriptType]	1	常量	识别描述符类型的常数，使用在有多个描述符的设备
10	[wDescriptorLength]	2	数字	描述符总长度，用于有多个描述符的设备

```
struct hid class descriptor {
    _u8  bDescriptorType;// 类别描述符的类型
    _u16 wDescriptorLength;// 报表描述符总长度
} attribute  ((packed));
struct hid_descriptor {
    _u8  bLength;// 描述符的字节数长度
    _u8  bDescriptorType;// 描述符类型（0x21）
    _u16 bcdHID;// HID 规范版本
    _u8  bCountryCode;// 国家识别号码
    _u8  bNumDescriptors;// 支持的附属描述符数目
    struct hid_class_descriptor desc[1];// 附属描述符
} _attribute_ ((packed));
```

5.2　Linux 下的 USB 驱动框架

Linux 操作系统中的 USB 主机驱动由三部分组成：USB 主机控制器驱动（HCD）、USB 驱动（USBD）和不同的 USB 设备类驱动，如图 5.5 所示。

1. USB 主机控制器驱动（HCD）

它是 USB 主机驱动程序中直接与硬件交互的软件模块，它的主要功能有：主机控制器硬件初始化；为 USBD 层提供相应的接口函数；提供根 Hub 设备配置、控制功能；完成 4 种类型的数据传输等。主控制器驱动程序能够更容易地将不同主控制器设备映射到 USB 系统中。因此客户可以在不知其设备连接哪个主控制器的情况下与设备相互作用。HCD 与 USBD 间的接口叫 HCDI，特定的 HCDI 由支持不同主控制器的操作系统定义。

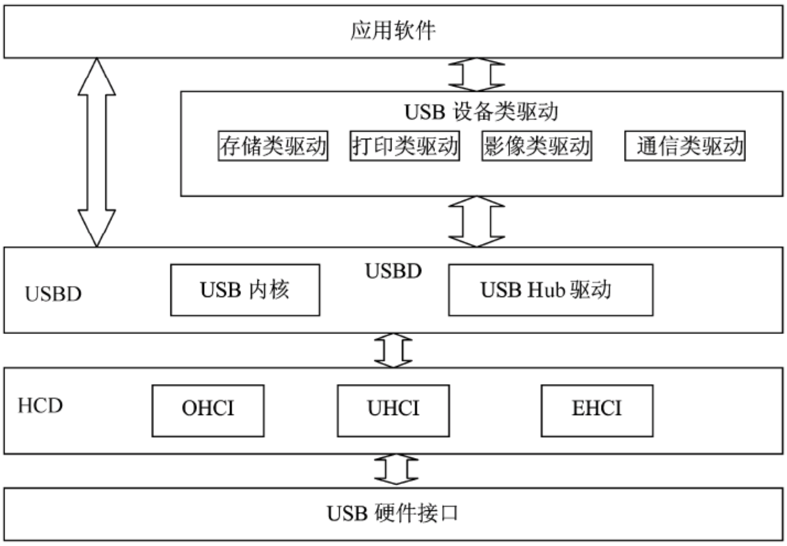


图 5.5　USB 驱动程序框架

2．USB 驱动（USBD）

UHCD 是 USB 主机驱动的核心，实现了与 USB 主控制器通信和控制 USB 主控制器的一些根本细节，并且它对系统软件的其他部分是隐蔽的。系统软件中的更高层通过 UHCD 的软件接口与主控制器通信。它的主要功能是 USB 总线管理、USB 的 Hub 驱动、设备类驱动接口、应用程序访问 USB 系统的文件接口。

USB 驱动程序（USBD）位于 UHCD 之上。它提供驱动器级的接口，满足现有设备驱动器设计的要求，USBD 所实现的准确细节随不同操作系统环境而有所不同，但 USBD 在不同操作系统环境下完成的是一样的工作。USBD 以 I/O 请求包（IRPs）的形式提供数据传输构架，它由通过特定管道（Pipe）传输数据的需求组成。此外，USBD 使客户端出现设备的一个抽象，以便于抽象和管理。作为抽象的一部分，USBD 拥有默认的管道。通过它可以访问所有的 USB 设备以进行标准的 USB 控制。该默认管道描述了一条 USBD 和 USB 设备间通信的逻辑通道。

3．USB 设备类驱动

USB 设备类驱动是与应用程序交互的软件模块，它主要实现特定的 USB 设备的访问、为应用程序提供访问接口等。USB 设备使用 USB 核心提供函数与设备通信，所以它应该是与平台无关的。Linux 内核支持的 USB 设备类包括：USB 打印设备、通信类设备、HID 设备类（HUMAN DEVICE CLASS）、存储设备类、语音设备类等。

USB 的支持在最近的内核开发周期中有了许多改进，其中最为显著的是新内核将支持 USB 2.0 设备。USB 2.0 是一种新的标准，支持设备带宽高达 480Mbps。支持此标准的设备通常被称作高速 USB 设备，它们正逐步占领市场。另外一个新的相关标准叫做 USB On-The-Go（或称作 USB OTG），它是 USB 协议中一个点到点的变种，用以直连设备；Linux 2.6 尚未支持它（Linux 2.6 的补丁是可以支持的）。除了设备支持外，多数 USB 设备的枚举方式都做了修正，使得 Linux 能访问现今许多同类型设备的所有实例（instance）。这一点对于大型打印机或存储设备来说相当有益

（虽然后者可能更倾向于使用专用存储总线）。很明显，这一领域的技术最近几年成长显著，Linux 对相关设备的支持也是紧跟市场的步伐。Linux 2.6 版本已经可以支持 USB 2.0 规范。在主机控制器方面，支持 USB 1.1 的 UHCI 与 OHCI 和 USB 2.0 的 EHCI。

Linux 中与 USB 驱动相关的重要结构包括 USB 驱动 usb_driver、USB 总线 usb_bus、USB 设备 usb_device。usb_driver 是 USB 驱动在 USB 核心中的标识，其中 probe、disconnect 成员函数为必须提供的函数，其他为可选的。关于/driver/usb 下的相关文件夹及其说明见表 5.5。

表 5.5　/driver/usb 下的文件夹

文件夹	说明
core	USB 驱动的核心模块，包括 urb、USB 文件系统、Hub 驱动
host	主机规范，包括 EHCI、UHCI、OHCI
storage	USB 接口的存储设备，如移动硬盘、U 盘驱动
input	HID 类设备
class	声音设备、打印机设备驱动
gadget	USB 从设备驱动

```
struct usb_driver {
    struct module *owner;
    const char *name;
    int (*probe) (struct usb_interface *intf,const struct usb_device_id *id);
    void (*disconnect) (struct usb_interface *intf);
    int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf);
    int (*suspend) (struct usb_interface *intf, u32 state);
    int (*resume) (struct usb_interface *intf);
    const struct usb_device_id *id_table;
    struct device driver driver;
};
int usb_register(struct usb_driver *);
void usb_deregister(struct usb_driver *);
```

USB 总线的结构如下：

```
struct usb_bus {
    struct device *controller;       // 主机硬件
    int busnum;                      // 总线数量
    char *bus_name;                  // 总线名称
    u8 otg_port;                     // OTG/HNP 端口数
    unsigned is b host:1;
    unsigned b_hnp_enable:1;         // OTG 主机是否允许 HNP
    int devnum next;
    struct usb_devmap devmap;        // 设备地址分配表
    struct usb_operations *op;       // USB 操作集合
    struct usb_device *root_hub;     // 根集线器
    struct list_head bus_list;       // 总线列表
    void *hcpriv;                    // 主机控制器私有数据
```

```
    int bandwidth_allocated;    // 总线上为周期性的传输（中断/同步）保留的平均带宽
    int bandwidth_int_reqs;                 // 中断传输带宽
    int bandwidth_isoc_reqs;                // 同步传输带宽
    struct dentry *usbfs_dentry;            // USB 文件系统路径
    struct dentry *usbdevfs_dentry;         // USB 设备文件系统路径
    struct class_device class_dev;          // 总线的类设备
    void (*release)(struct usb_bus *bus);   // 总线内存释放函数
};
```

struct usb_device 描述一个具体的 USB 设备：

```
struct usb device {
    int     devnum;                     // 在 USB 总线上的地址
    char    devpath [16];               // 设备路径
    enum usb_device_state state;        // 设备状态
    enum usb_device_speed speed;        // 设备速率（高速/全速/低速）
    struct usb_tt   *tt;                // 全速/低速设备，或高速 HUB
    int ttport;                         // 在 tt hub 上的端口
    struct semaphore serialize;
    unsigned int toggle[2];             // 端点的方向，0=输入；1=输出
    int epmaxpacketin[16];              // 输入端点最大包数
    int epmaxpacketout[16];             // 输出端点最大包数
    struct usb_device *parent;          // 设备连接的 HUB
    struct usb_bus *bus;                // 隶属的总线
    struct device dev;                  // 常规设备接口
    struct usb_device_descriptor descriptor;        // 设备描述符
    struct usb_host_config *config;                 // 所有的配置
    struct usb_host_config *actconfig;              // 激活的配置
    char **rawdescriptors;                          // 每个配置的原始描述符
    int have_langid;                                // 是否 string_langid 域合法
    int string_langid;                              // 字符串的语言 ID
    void *hcpriv;                                   // 主机控制器私有数据
    struct list head filelist;
    struct dentry *usbfs_dentry;                    // USB 文件系统路径
    struct dentry *usbdevfs_dentry;                 // USB 设备文件系统路径
    int maxchild;                                   // 端点的数量
    struct usb_device *children[USB_MAXCHILDREN];   // 包含的 USB 子设备
};
```

5.3　USB 请求块 urb

USB 系统基于消息传递机制。消息被称作 urb，即 USB 请求块。USB 驱动一般通过一个 USB 请求块 urb 与 USB 设备通信，它通过调用 usb_submit_urb 发送 USB 请求块。urb 被用来以一种异步的方式进行传输。usb_submit_urb 是个异步调用，它将立即返回。urb 被放入一个队列，处理完

毕后返回到一个 completion 处理函数。completion 处理函数是 urb 结构的一个成员。在这个函数里可以检查 urb -> status 来确定是否有错误发生。

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
```

注意 mem_flags 有 GFP_KERNEL、GFP_NOFS、GFP_NOIO 和 GFP_ATOMIC 等选择。GFP_NOFS 还没有被实现，GFP_ATOMIC 用在一个完成处理函数、中断、工作队列和定时器中，或获得了 spinlock 和 rwlock 的情况。GFP_NOIO 用于阻塞 I/O 路径或存储设备的错误处理。其他情况都用 GFP_KERNEL。

要取消挂起的请求，可以用 usb_unlink_urb 函数。completion 处理函数：

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
```

completion 处理函数是在中断上下文中调用的，它用在下面三种情况：

（1）数据传输成功，urb->status == 0。

（2）传输过程发生错误。

（3）USB 核心取消了 urb。

在 completion 处理函数中注意：

（1）不要使用任何可能睡眠的函数。

（2）处理过程尽量快速。

（3）必须检查 urb->status。

图 5.6 显示了 urb 的全部生命过程。urb 包含如下成员：

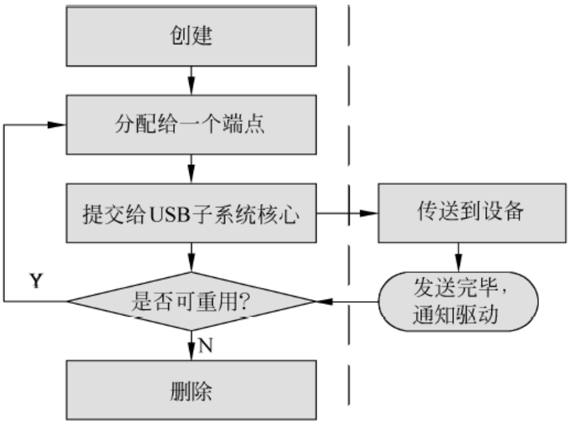


图 5.6 urb 结构的生命期

```
struct urb
{
    //以下为私有，只对 USB 核心和主机控制器可见
    struct kref kref;            // urb 引用计数
    spinlock_t lock;             // urb 锁
    void *hcpriv;                // 主机控制器的私有数据
    struct list_head urb_list;  // 活动的 urb 列表指针
    int bandwidth;               // INT/ISO 请求的带宽
    atomic_t use_count;          //并发提交计数
    u8 reject;
   //以下为公共的，可以被驱动访问
    struct usb_device *dev;      //urb 关联的设备
    unsigned int pipe;           //管道信息
    int status;                  //状态
    unsigned int transfer_flags;//处理 urb 的一些设置
    void *transfer_buffer;       //数据缓冲
    dma_addr_t transfer_dma;     //DMA 传输用的数据缓冲
    int transfer_buffer_length; //数据长度
    int actual_length;           //数据收发的实际长度
```

```
    unsigned char *setup_packet;    //控制传输过程专有的建立包
    dma_addr_t setup_dma;           //DMA 方式下的建立包
    int start_frame;                //初始帧号
    int number_of_packets;          //同步包的数量
    int interval;                   //传输间隔
    int error_count;                //同步传输的错误数量
    void *context;                  //完成处理函数用到的数据
    usb_complete_t complete;        //完成处理函数
    struct usb iso packet descriptor iso frame desc[0];
    //允许单个 urb 一次定义多个同步传输
};
```

初始化 urb 结构中的 Pipe 的方法有：

```
#define usb_sndctrlpipe(dev,endpoint)   \
((PIPE_CONTROL << 30) | _create_pipe(dev,endpoint))
#define usb_rcvctrlpipe(dev,endpoint)   \
((PIPE_CONTROL << 30) | _create_pipe(dev,endpoint) | USB_DIR_IN)
//以上为控制 urb 的管道
#define usb sndisocpipe(dev,endpoint)  \
((PIPE ISOCHRONOUS << 30) |  create pipe(dev,endpoint))
#define usb rcvisocpipe(dev,endpoint)  \
((PIPE ISOCHRONOUS << 30) |  create pipe(dev,endpoint) | USB DIR IN)
//以上为异步 urb 的管道
#define usb_sndbulkpipe(dev,endpoint)  \
((PIPE_BULK << 30) | _create_pipe(dev,endpoint))
#define usb_rcvbulkpipe(dev,endpoint)  \
((PIPE_BULK << 30) | _create_pipe(dev,endpoint) | USB_DIR_IN)
//以上为批量 urb 的管道
#define usb sndintpipe(dev,endpoint)  \
((PIPE INTERRUPT << 30) |  create pipe(dev,endpoint))
#define usb rcvintpipe(dev,endpoint)  \
((PIPE INTERRUPT << 30) |  create pipe(dev,endpoint) | USB DIR IN)
//以上为中断 urb 的管道
```

urb 相关的操作函数包括：

```
//创建一个 urb:
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
//iso_packets 代表等时数据包的数量，如果不创建等时 urb 则该值为 0。mem_flags 为创建标志
//销毁 urb:
void usb_free_urb(struct urb *urb) );
//填充中断 urb:
static inline void usb_fill_int_urb (struct urb *urb,struct usb_device *dev,unsigned
int pipe,void *transfer_buffer,int buffer_length,usb_complete_t complete,void
*context,int interval) );
```

```
//填充批量 urb:
static inline void usb_fill_bulk_urb (struct urb *urb,struct usb_device
*dev,unsigned int pipe,void *transfer_buffer,int buffer_length,usb_complete_t
complete,void *context) );
//填充控制 urb:
static inline void usb_fill_control_urb (struct urb *urb,struct usb_device
*dev,unsigned int pipe,unsigned char *setup_packet,void *transfer_buffer,int
buffer_length,usb_complete_t complete,void *context) );
//提交 urb:
int usb_submit_urb(struct urb *urb, int mem_flags) );
//取消 urb:
int usb_unlink_urb(struct urb *urb) );
void usb_kill_urb(struct urb *urb) );
```

等时 urb 没有专门的填充函数，可以通过下面的方式提交等时 urb：

```
struct urb *urb = uvd->sbuf[i].urb;
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb rcvisocpipe(dev, uvd->video endp);
urb->interval = 1;
urb->transfer flags = URB ISO ASAP;
urb->transfer buffer = uvd->sbuf[i].data;
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = pktsz * FRAMES_PER_DESC;
for (j=k=0; j < FRAMES_PER_DESC; j++, k += pktsz) {
    urb->iso_frame_desc[j].offset = k;
    urb->iso_frame_desc[j].length = pktsz;
}
```

有时 USB 驱动程序不采用 urb：

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int
len, int *actual_length, int timeout)
//封装一个批量 urb，发送后等待完成
int usb_control_msg(struct usb_device *dev, unsigned int pipe, _u8 request, _u8
requesttype,_u16 value, _u16 index, void *data, _u16 size, int timeout)
//封装一个控制 urb，发送后等待完成
```

如果程序中用到了 DMA，可以用下面的函数分配 DMA 缓冲：

```
void *usb_buffer_alloc (struct usb_device *dev, size_t size,gfp_t mem_flags,
dma_addr_t *dma);
//分配 DMA 缓冲。dev 代表关联的设备; size 是缓冲尺寸; mem_flags 是 kmalloc 标志; dma 是 DMA
//地址。返回分配的 CPU 地址空间指针
void usb_buffer_free (struct usb_device *dev, size_t size,void *addr, dma_addr_
```

```
t dma);
//释放 DMA 缓冲。dev 代表关联的设备; size 是缓冲尺寸; addr 是上面函数分配的 CPU 地址指针; dma
//是 DMA 地址
```

另外需要设置 DMA 标志，这个表示对于建立包是 URB_NO_SETUP_DMA_MAP，对于数据包是 URB_NO_TRANSFER_DMA_MAP。例如：

```
buf = usb_buffer_alloc(dev->udev,count, GFP_KERNEL,&urb->transfer_dma);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

5.4 USB 骨架程序

USB 骨架程序（usb-skeleton）是 USB 驱动程序的基础，通过对它源码的学习和理解，可以迅速地了解 USB 驱动架构，迅速地开发 USB 硬件的驱动。USB 骨架程序在 Linux 内核源码目录中的 driver/usb/usb-skeleton.c 中。通过它仅需要修改极少的部分，就可以完成一个 USB 设备的驱动。

如果想写一个 Linux 驱动程序，首先要熟悉 USB 协议规范。USB 主页上有它的帮助。一些比较典型的驱动可以在上面发现，同时还有上面介绍的 USB urb 的概念，这个是 USB 驱动程序中最基本的。Linux USB 驱动程序需要做的第一件事情就是在 Linux USB 子系统里注册，并提供一些相关信息，例如这个驱动程序支持哪种设备，当被支持的设备从系统插入或拔出时，会有哪些动作。所有这些信息都传送到 USB 子系统中。

本书以 Linux 2.6.20 中 USB 骨架驱动程序为例讲解开发 USB 驱动的基本方法。该驱动用 struct usb_skel 包容了所有需要的内容。

```
struct usb skel {
    struct usb_device   *dev;                // USB 设备驱动
    struct usb_interface *interface;         // 设备的接口
    struct semaphore    limit_sem;           // 限制进行中的写操作数量
    unsigned char     *bulk_in_buffer;       // 数据接收缓冲
    size_t  bulk_in_size;                    // 数据接收缓冲大小
    _u8     bulk_in_endpointAddr;            // 批量输入端点地址
    _u8     bulk_out_endpointAddr;           // 批量输出端点地址
    struct kref kref;
    struct mutex        io_mutex;            // 同步变量
};
```

skel_driver 是一个 USB 驱动的实例：

```
static struct usb_driver skel_driver = {
    .name = "skeleton",
    .probe = skel_probe,
    .disconnect = skel_disconnect,
```

```
    .id_table = skel_table,
};
```

变量 name 是一个字符串，它对驱动程序进行描述。probe 和 disconnect 是函数指针，当设备与在 id_table 中变量信息匹配时，此函数被调用。

USB 驱动程序在注册时会发送一个命令给 usb_register，通常在驱动程序的初始化函数里。当要从系统卸载驱动程序时，需要注销 USB 子系统。即需要 usb_unregister 函数处理：

```
static int _init usb_skel_init(void)
{
    int result;
    //向 USB 子系统注册这个驱动
    result = usb_register(&skel_driver);
    if (result)
        err("usb register failed. Error number %d", result);
    return result;
}
static void  exit usb skel exit(void)
{
    //在 USB 子系统中注销这个驱动
    usb_deregister(&skel_driver);
}
```

当 USB 设备插入时，为了使 Linux-hotplug（Linux 中 PCI、USB 等设备热插拔支持）系统自动装载驱动程序，需要创建一个 MODULE_DEVICE_TABLE。代码如下（这个模块仅支持某一特定设备）：

```
//设备的厂商 ID
#define USB SKEL VENDOR ID  0xFFF0
#define USB_SKEL_PRODUCT_ID 0xFFF0
//使用本驱动的设备列表
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { } // 终止项目
};
//描述特定的驱动支持的设备
MODULE_DEVICE_TABLE (usb, skel_table);
```

USB_DEVICE 宏利用厂商 ID 和产品 ID 提供了一个设备的唯一标识。当系统插入一个与 ID 匹配的 USB 设备到 USB 总线时，驱动会在 USB core 中注册。驱动程序中的 probe 函数也就会被调用。usb_device 结构指针、接口号和接口 ID 都会被传递到函数中。

```
static int skel_probe(struct usb_interface *interface, const struct usb_device_
id *id)
{
    struct usb_skel *dev;
```

```
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        size_t buffer_size;
        int i;
        int retval = -ENOMEM;
        //为设备驱动分配内存并初始化
        dev = kzalloc(sizeof(*dev), GFP_KERNEL);
        if (!dev) {
            err("Out of memory");
            goto error;
        }
        kref_init(&dev->kref);//初始化引用计数
        sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
        mutex_init(&dev->io_mutex);
        //根据接口获取设备
        dev->udev = usb_get_dev(interface_to_usbdev(interface));
        dev->interface = interface;
        //建立端点信息，使用第一个批量输入、输出端点
        iface desc = interface->cur altsetting;
        for (i = 0; i < iface desc->desc.bNumEndpoints; ++i) {
            endpoint = &iface desc->endpoint[i].desc;
            if (!dev->bulk in endpointAddr &&
                usb endpoint is bulk in(endpoint)) {
                //找到一个批量输入端点
                buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
                dev->bulk_in_size = buffer_size;
                dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
                dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
                if (!dev->bulk_in_buffer) {
                    err("Could not allocate bulk_in_buffer");
                    goto error;
                }
            }
            if (!dev->bulk out endpointAddr &&
               usb endpoint is bulk out(endpoint)) {
                //找到一个批量输出端点
                dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
            }
        }
        //没有找到可用端点
        if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
            err("Could not find both bulk-in and bulk-out endpoints");
            goto error;
        }
        //将设备信息保存到接口结构中
```

```
    usb_set_intfdata(interface, dev);
    //一切就绪，注册次设备
    retval = usb_register_dev(interface, &skel_class);
    if (retval) {//注册失败
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
    }
    info("USB Skeleton device now attached to USBSkel-%d", interface->minor);
    return 0;
    error:
    if (dev)
        kref_put(&dev->kref, skel_delete);
    return retval;
}
```

大多 USB 驱动程序都会涉及另外一个驱动系统，例如 SCSI，网络或者 TTY 子系统。这些驱动程序在其他驱动系统中注册，同时任何用户空间的交互操作通过那些接口提供，比如把 SCSI 设备驱动作为 USB 驱动所涉及的另外一个驱动系统，那么 USB 设备的 read、write 等操作，就相应按 SCSI 设备的 read、write 函数进行访问。但是对于扫描仪等驱动程序来说，并没有一个匹配的驱动系统可以使用，那就要处理与用户空间的 read、write 等交互函数。USB 子系统提供一种方法去注册一个次设备号和 file_operations 函数指针，这样就可以与用户空间实现方便地交互。这个方法就是 usb_class_driver，它提供了次设备号、文件接口、devfs 接口等。

```
static const struct file_operations skel_fops = {
    .owner = THIS_MODULE,
    .read = skel_read,
    .write = skel_write,
    .open = skel_open,
    .release = skel_release,
};
static struct usb_class_driver skel_class = {
    .name = "skel%d",
    .fops = &skel_fops,
    .minor_base = USB_SKEL_MINOR_BASE,
};
int usb_register_dev(struct usb_interface *intf,struct usb_class_driver
*class_driver);
```

下面是设备断开处理函数：

```
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;
```

```
        //避免 skel_open 与 skel_disconnect 发生资源抢占
        lock_kernel();
        dev = usb_get_intfdata(interface);
        usb_set_intfdata(interface, NULL);
        //注销次设备
        usb_deregister_dev(interface, &skel_class);
        //防止更多的 I/O 操作
        mutex lock(&dev->io mutex);
        dev->interface = NULL;
        mutex unlock(&dev->io mutex);
        unlock kernel();
        //减少引用计数
        kref_put(&dev->kref, skel_delete);
        info("USB Skeleton #%d now disconnected", minor);
    }
```

现在，skeleton 驱动就已经和设备绑定上了，任何用户态程序要操作此设备都可以通过 file_operations 结构所定义的函数进行了。首先要 open 此设备。

```
    static int skel_open(struct inode *inode, struct file *file)
    {
        struct usb_skel *dev;
        struct usb_interface *interface;
        int subminor;
        int retval = 0;
        //得到次设备号
        subminor = iminor(inode);
        //通过次设备号寻找接口
        interface = usb find interface(&skel driver, subminor);
        if (!interface) {
            err ("%s -error, can't find device for minor %d",
                _FUNCTION_, subminor);
            retval = -ENODEV;
            goto exit;
        }
        //得到用 usb_set_intfdata 保存的设备
        dev = usb get intfdata(interface);
        if (!dev) {
            retval = -ENODEV;
            goto exit;
        }
        //防止设备自动挂起
        retval = usb_autopm_get_interface(interface);
        if (retval)
            goto exit;
```

```
    //增加引用计数
    kref_get(&dev->kref);
    //保存设备到文件私有数据
    file->private_data = dev;
exit:
    return retval;
}
```

读函数先发起一个读操作，从设备中读出数据。如果成功，则将数据复制到应用层：

```
Static ssize_t skel_read(struct file *file, char *buffer, size_t count, loff_
t *ppos)
{
    struct usb_skel *dev;
    int retval;
    int bytes_read;
   //从私有数据中获取设备信息
    dev = (struct usb_skel *)file->private_data;
   //获得锁
    mutex lock(&dev->io mutex);
    if (!dev->interface) {
        retval = -ENODEV;
        goto exit;
    }
    //发布一个阻塞式批量读操作
    retval = usb_bulk_msg(dev->udev,
                  usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                  dev->bulk_in_buffer,
                  min(dev->bulk_in_size, count),
                  &bytes read, 10000);
    //将数据复制到应用层
    if (!retval) {
        if (copy to user(buffer, dev->bulk in buffer, bytes read))
            retval = -EFAULT;
        else
            retval = bytes_read;
    }
exit:
    //释放锁
    mutex_unlock(&dev->io_mutex);
    return retval;
}
```

write 函数先分配一个 urb，并把用户空间的数据复制到驱动中，然后填充到刚分配的批量 urb 结构，最后向 USB 子系统提交该 urb：

```
Static ssize_t skel_write(struct file *file, const char *user_buffer, size_t count,
loff_t *ppos)
{
    struct usb_skel *dev;
    int retval = 0;
    struct urb *urb = NULL;
    char *buf = NULL;
    size_t writesize = min(count, (size_t)MAX_TRANSFER);
    //从私有数据中获取设备信息
    dev = (struct usb_skel *)file->private_data;
    //确保有数据要传输
    if (count == 0)goto exit;
    //限制 urb 的数量，防止 RAM 用完
    if (down_interruptible(&dev->limit_sem)) {
        retval = -ERESTARTSYS;
        goto exit;
    }
    //获得锁
    mutex_lock(&dev->io_mutex);
    if (!dev->interface) {      // disconnect()被唤醒
        retval = -ENODEV;
        goto error;
    }
    //创建 urb
    urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!urb) {
        retval = -ENOMEM;
        goto error;
    }
    //分配数据缓冲
    buf = usb_buffer_alloc(dev->udev, writesize, GFP_KERNEL, &urb->transfer_dma);
    if (!buf) {
        retval = -ENOMEM;
        goto error;
    }
    //将用户数据复制到内核
    if (copy_from_user(buf, user_buffer, writesize)) {
        retval = -EFAULT;
        goto error;
    }
    //填充 urb
    usb_fill_bulk_urb(urb, dev->udev,
              usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
              buf, writesize, skel_write_bulk_callback, dev);
    urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

```
    //提交 urb 块
    retval = usb_submit_urb(urb, GFP_KERNEL);
    if (retval) {
        err("%s-failed submitting write urb, error %d", _FUNCTION_, retval);
        goto error;
    }
    //释放 urb
    usb free urb(urb);
    //解锁
    mutex unlock(&dev->io mutex);
    return writesize;
error:
    if (urb) {
        usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
        usb_free_urb(urb);
    }
    mutex_unlock(&dev->io_mutex);
    up(&dev->limit_sem);
exit:
    return retval;
}
```

回调函数主要是释放内存：

```
static void skel_write_bulk_callback(struct urb *urb)
{
    struct usb_skel *dev;
    dev = (struct usb_skel *)urb->context;
    if (urb->status &&
        !(urb->status == -ENOENT ||
         urb->status == -ECONNRESET ||
         urb->status == -ESHUTDOWN)) {
        err("%s - nonzero write bulk status received: %d",
            FUNCTION , urb->status);
    }
    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
            urb->transfer_buffer, urb->transfer_dma);
    up(&dev->limit_sem);
}
```

最后是释放函数：

```
static int skel_release(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    dev = (struct usb_skel *)file->private_data;
```

```
        if (dev == NULL)
            return -ENODEV;
        //允许设备自挂起
        mutex_lock(&dev->io_mutex);
        if (dev->interface)
            usb_autopm_put_interface(dev->interface);
        mutex_unlock(&dev->io_mutex);
        //减少引用计数
        kref_put(&dev->kref, skel_delete);
        return 0;
    }
```

5.5 USB 文件系统

通过/proc/bus/usb/devices 文件的内容，就可以获得连接的设备信息，包括设备标识和制造商标等信息。USB 文件系统中的符号意义如表 5.6 所示。

表 5.6 USB 文件系统中的符号意义

符号	含义
T	总线拓扑结构（Lev，Prnt，Port，Cnt），指 USB 设备和主机之间的连接方式
B	带宽（仅用于 USB 主控制器）
D	设备描述信息
P	产品标识信息
S	串描述符
C	配置描述信息
I	接口描述信息
E	终端点描述信息

用 d 表示十进制数，x 表示十六进制数，s 表示字符串，逐一介绍各种信息的格式。

1. 拓扑信息

T: Bus=dd Lev=dd Prnt=dd Port=dd Cnt=dd Dev#=ddd Spd=ddd MxCh=dd

依次是拓扑信息标志、总线编号、此总线在拓扑结构中的层次、父设备号、此设备的父连接器/端口、该层的设备数、设备编号、设备速度（Mbps）、最大子设备。

2. 带宽信息

B: Alloc=ddd/ddd us (xx%), #Int=ddd, #Iso=ddd

依次是带宽信息标志、分配给此总线的总带宽、中断请求号、同步请求编号。

3. 设备描述信息和产品标识信息

D: Ver=x.xx Cls=xx(s) Sub=xx Prot=xx MxPS=dd #Cfgs=dd

依次是设备信息标志 D、设备 USB 版本、设备类型、设备子类型、设备协议默认终端点的最大包尺寸、配置编号。

4．产品标识信息

P: Vendor=xxxx ProdID=xxxx Rev=xx.xx

包括制造商标识编码、产品标识编码、产品修订号。

5．串描述信息

S: Manufacturer=ssss

串描述信息、设备上读出的制造商信息。

S: Product=ssss

串描述信息、设备上读出的产品描述信息。

S: SerialNumber=ssss

串描述信息、设备上读出的序列号。

6．配置描述信息

C: #Ifs=dd Cfg#=dd Atr=xx MPwr=dddmA

配置信息标志、接口数、配置编号、属性、最大电流（mA）。

7．接口描述信息

I: If#=dd Alt=dd #EPs=dd Cls=xx(sssss) Sub=xx Prot=xx Driver=ssss

一个 USB 设备可有多个接口描述信息。接口描述信息包括接口信息标志、接口编号、可变设置编号、中断点数、接口类、接口子类、接口协议、驱动名称。

8．终端点信息

E: Ad=xx(s) Atr=xx(ssss) MxPS=dddd Ivl=dddms

终端点信息包括终端点信息标志、终端点地址（I=In，O=Out）、属性（终端点类型）、终端点最大包尺寸、间隔。

5.6 USB 摄像头驱动

5.6.1 USB 摄像头原理

USB 摄像头的工作原理大致为：景物通过镜头（LENS）生成的光学图像投射到图像传感器表面上，然后转为电信号，经过 A/D（模数转换）转换后变为数字图像信号，再送到处理芯片中存储和加工处理，再通过 USB 接口发送到 USB 主机。图 5.7 是典型的 USB 摄像头芯片 OV511+芯片的结构图，可以作为 USB 摄像头处理芯片的代表。OV511+包含一个数字摄像头接口、DRAM 接口和一个 USB 设备控制器。OV511+包含两个端点，端点 0 支持控制传输，用来传送控制信息，

端点 1 支持同步传输，用来传送视频数据。OV511+可支持 8 种不同的同步传输接口，速度达 7.5Mbps。

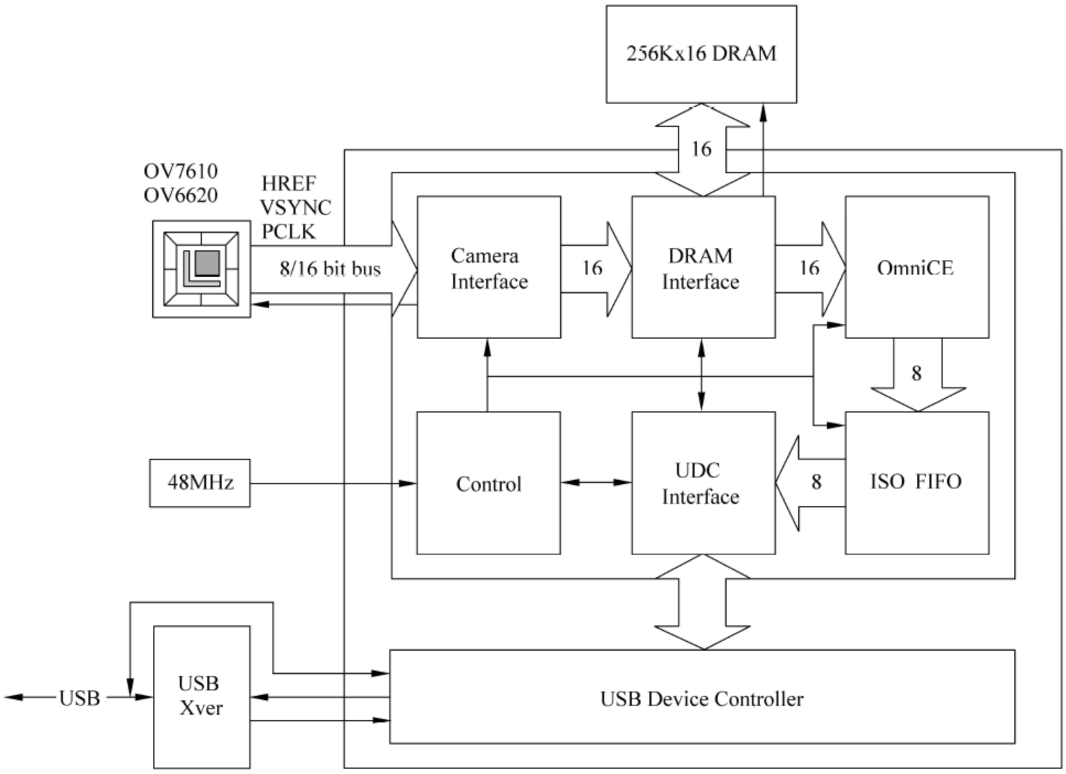


图 5.7 OV511+结构原理

5.6.2 Video4Linux 规范

摄像头属于视频类设备。在目前的 Linux 核心中，视频部分的标准是 Video for Linux（简称 V4L）。这个标准其实定义了一套接口，内核、驱动、应用程序以这个接口为标准进行交流。目前的 V4L 涵盖了视、音频流捕捉及处理等内容，USB 摄像头也属于它支持的范畴。USB 摄像头驱动首先在内核中声明一个 video_device 结构，并为其指定文件操作函数指针数组 fops，向系统注册。在应用程序发出文件操作的相关命令时，核心根据这些指针调用相应函数，并将该结构作为参数传递给它们。这样，就完成了驱动和核心之间的通信。

在 Linux 中，视频设备是设备文件，可以像访问普通文件一样对其进行读写，摄像头文件一般是/dev/v4l/video。在进行视频捕捉之前，需要做一些必要的设置工作。这些设置涉及到如下结构：

```
struct video picture
{
    _u16    brightness;//亮度
    _u16    hue;
```

```
     u16     colour;
    _u16     contrast;//对比度
    _u16     whiteness;//黑色或白色
    _u16     depth; //颜色深度
    _u16   palette;//调色板
};
struct video_window
{
    _u32     x,y;     //窗体位置
    _u32     width,height;//窗体尺寸
     u32     chromakey;
     u32     flags;
    struct  video_clip _user *clips;
    int clipcount;
#define VIDEO_WINDOW_INTERLACE  1
#define VIDEO_WINDOW_CHROMAKEY 16
#define VIDEO_CLIP_BITMAP   -1
#define VIDEO_CLIPMAP_SIZE  (128 * 625)
};
struct video capture
{
    _u32     x,y;     //图像偏移
    _u32     width, height;  //捕捉区域
     u16     decimation; //采集间隔
    _u16     flags;  //采集标志
#define VIDEO_CAPTURE_ODD       0
#define VIDEO_CAPTURE_EVEN      1
};
```

Video4Linux 下视频编程的流程很简单，首先打开视频设备文件：

```
Int fd= open("/dev/v4l/video0", O RDONLY);
if(fd<0) return -1;
```

接着对摄像参数进行设置。一般是先通过 I/O 控制命令读取设备信息，然后对特定项进行修改，最后通过 I/O 控制命令保存到内核中。

```
struct video picture vp;
struct video window vw;
struct video capture vcp;
if(ioctl(fd, VIDIOCGCAP, &vc)<0)
        printf("VIDIOCGCAP fail\n");
    else
        printf("max width %d, height %d\nmin width %d, height %d\n",
            vc.maxwidth, vc.maxheight, vc.minwidth, vc.minheight);
   //设置捕捉的宽度与高度
```

```
    vw.width = 240;
    vw.height = 320;
    if(ioctl(fd, VIDIOCSWIN, &vw)<0)
        printf("VIDIOCSWIN fail\n");
    //验证设置
    if(ioctl(fd, VIDIOCGWIN, &vw)<0)
        printf("VIDIOCGWIN fail\n");
    else
        printf("current width %d, height %d\n",vw.width, vw.height);
    vcp.width = vw.width;
    vcp.height = vw.height;
    if(ioctl(fd, VIDIOCSCAPTURE, &vcp)<0)
        printf("VIDIOCSCAPTURE fail\n");
    else {
        if(ioctl(fd, VIDIOCGCAPTURE, &vcp)<0)
            printf("VIDIOCGCAPTURE fail\n");
        else
            printf("capture width %d, height %d\n", vcp.width, vcp.height);
    }
    //设置调色板
    vp.palette = VIDEO_PALETTE_YUV422;
    if(ioctl(fd, VIDIOCSPICT, &vp)<0)
        printf("VIDIOCSPICT fail\n");
    //验证设置
    if(ioctl(fd, VIDIOCGPICT, &vp)<0)
        printf("VIDIOCGPICT fail\n");
    else
        printf("current palette %d\n", vp.palette);
```

视频采集的第一种方法：通过映射得到视频驱动的数据缓冲，然后直接对映射后的缓冲进行读操作。

```
if(ioctl(fd, VIDIOCGMBUF, &vm)<0) {
    printf("VIDIOCGMBUF fail\n");
    mmap camera = 0;
} else {
    printf("current camera buffer size %d, total frames %d\n",
        vm.size, vm.frames);
    //内存映射
    buf = (_u8 *)mmap(0, vm.size, PROT_READ, MAP_SHARED, fd, 0);
    if((int)buf==-1) {
        printf("mmap camera fail!\n");
        mmap camera = 0;
    } else
        puts("mmap camera ok.\n");
}
```

捕捉流程就是不断调用 VIDIOCCAPTURE 控制：

```
fd_set rfds;
FD_ZERO(&rfds);
while(1)
{
        //发送读图片命令
        if(ioctl(fd, VIDIOCCAPTURE, STILL_IMAGE)<0)
        {
                printf("VIDIOCCAPTURE fail\n");
                break;
        }
    FD_SET(0, &rfds);
    FD_SET(fd, &rfds);
    tv.tv_sec = 3;
    tv.tv_usec = 0;
    //等待图像数据
    select(fd+1, &rfds, NULL,  NULL, &tv);
    if(FD_ISSET(fd, &rfds))
    {
        if(mmap_camera) {
            i = read(fd, buf, 0);
            if(i<0)
            {
                printf("read fail!%d\n", i);
                break;
            }
    }
    else
    {
        printf("wait timeout, break...\n");
        break;
    }
    //这里可以显示图像
    display(buf);
}
//取消映射
munmap(buf, vm.size);
free(buf);
close(fd);
```

视频采集的第二种方法：直接读设备，即调用 read 函数。

```
buf = malloc(image_width*image_height*2);
if(!buf) {
        printf("fail to allocate memory for camera!\n");
```

```
            close(fd);
            munmap(fb_buf, screensize);
            close(fbfd);
            return -1;
    }
    i = read(fd, buf, image_width*image_height*2);
    if(i<0) {
            fprintf(stderr, "read fail! %d\n", i);
            break;
    }
```

5.6.3　OV511 驱动分析与编译

Linux 内核中自带 OV511 芯片的 USB 摄像头驱动。OV511 芯片主要包括 CAMERA 接口、DRMA 接口、ISO FIFO 接口和 OmniCE 以及 USB 控制器等部分。

```
static struct usb_driver ov511_driver;
static struct usb_device_id device_table [] = {
    { USB_DEVICE(VEND_OMNIVISION, PROD_OV511) },
    { USB_DEVICE(VEND_OMNIVISION, PROD_OV511PLUS) },
    { USB_DEVICE(VEND_OMNIVISION, PROD_OV518) },
    { USB_DEVICE(VEND_OMNIVISION, PROD_OV518PLUS) },
    { USB_DEVICE(VEND_MATTEL, PROD_ME2CAM) },
    { }  // 终止项目
};
//加入到 USB 设备列表
MODULE_DEVICE_TABLE (usb, device_table);
```

驱动访问 OV511 的寄存器通过 usb_control_msg。注意 usb_control_msg 的标志位参数（第 4 个）如果设置为 USB_DIR_IN，则是从设备到主机，如果没有设置或设置为 USB_DIR_OUT，则通信方向是从主机到设备。

```
// 写 OV51x 寄存器
static int reg_w(struct usb_ov511 *ov, unsigned char reg, unsigned char value)
{
    int rc;
    PDEBUG(5, "0x%02X:0x%02X", reg, value);
    //获取锁
    down(&ov->cbuf_lock);
    ov->cbuf[0] = value;
    //通过 USB 控制命令发送数据。这里没有表明方面，默认为 USB_DIR_OUT
    rc = usb_control_msg(ov->dev,
                usb_sndctrlpipe(ov->dev, 0),
                (ov->bclass == BCL_OV518)?1:2 // REG_IO,
```

```
                    USB TYPE VENDOR | USB RECIP DEVICE,
                    0, (_u16)reg, &ov->cbuf[0], 1, HZ);
    //解锁
    up(&ov->cbuf_lock);
    if (rc < 0)
        err("reg write: error %d: %s", rc, symbolic(urb_errlist, rc));
    return rc;
}
// 读 OV51x 寄存器
// 返回值: 负数错误，正数或零为所取数据
static int reg_r(struct usb_ov511 *ov, unsigned char reg)
{
    int rc;
    //获取锁
    down(&ov->cbuf_lock);
    //这里标志位设置成 USB_DIR_IN
    rc = usb_control_msg(ov->dev,
                    usb_rcvctrlpipe(ov->dev, 0),
                    (ov->bclass == BCL OV518)?1:3, // REG IO
                    USB DIR IN | USB TYPE VENDOR | USB RECIP DEVICE,
                    0, ( u16)reg, &ov->cbuf[0], 1, HZ);
    if (rc < 0) {
        err("reg read: error %d: %s", rc, symbolic(urb errlist, rc));
    } else {
        rc = ov->cbuf[0];
        PDEBUG(5, "0x%02X:0x%02X", reg, ov->cbuf[0]);
    }
    //解锁
    up(&ov->cbuf_lock);
    return rc;
}
```

OV511 驱动中的 mmap 接口通过 remap_page_range 实现：

```
static int  ov51x_v4l1_mmap(struct file *file, struct vm_area_struct *vma)
{
    struct video_device *vdev = file->private_data;
    unsigned long start = vma->vm_start;
    unsigned long size  = vma->vm_end-vma->vm_start;
    struct usb_ov511 *ov = video_get_drvdata(vdev);
    unsigned long page, pos;
    if (ov->dev == NULL)return -EIO;
    PDEBUG(4, "mmap: %ld (%lX) bytes", size, size);
    //判断尺寸
    if (size > (((OV511_NUMFRAMES* MAX_DATA_SIZE(ov->maxwidth,
                        ov->maxheight)+ PAGE_SIZE - 1) & ~(PAGE_SIZE - 1)))
```

```
            return -EINVAL;
        //获取锁
        if (down_interruptible(&ov->lock))
            return -EINTR;
        //按页进行映射
        pos = (unsigned long)ov->fbuf;
        while (size > 0) {
            page = kvirt_to_pa(pos);
            if (remap_page_range(vma, start, page, PAGE_SIZE,PAGE_SHARED)) {
            up(&ov->lock);
                return -EAGAIN;
            }
            start += PAGE_SIZE;
            pos += PAGE_SIZE;
            if (size > PAGE_SIZE)
                size -= PAGE_SIZE;
            else
                size = 0;
        }
        up(&ov->lock); //解锁
        return 0;
    }
```

那么驱动是如何采集图像的呢？在驱动程序初始化的过程中，分配一个 urb，填充等时 urb，并定义了完成处理函数 ov51x_isoc_irq。

```
    ov51x_init_isoc(struct usb_ov511 *ov)
    {
        ...
        //计算包尺寸
        if (ov->bclass == BCL_OV518) {
            if (packetsize == -1) {
                ov518_set_packet_size(ov, 640);
            } else {
                info("Forcing packet size to %d", packetsize);
                ov518_set_packet_size(ov, packetsize);
            }
        } else {
            if (packetsize == -1) {
                ov511_set_packet_size(ov, size);
            } else {
                info("Forcing packet size to %d", packetsize);
                ov511_set_packet_size(ov, packetsize);
            }
        }
```

```
    //填充了等时 urb
    for (n = 0; n < OV511_NUMSBUF; n++) {
        urb = usb_alloc_urb(FRAMES_PER_DESC, GFP_KERNEL);
        if (!urb) {
            err("init isoc: usb_alloc_urb ret. NULL");
            return -ENOMEM;
        }
        ov->sbuf[n].urb = urb;
        urb->dev = ov->dev;
        urb->context = &ov->sbuf[n];
        urb->pipe = usb_rcvisocpipe(ov->dev, OV511_ENDPOINT_ADDRESS);
        //设置等时传输方式
        urb->transfer_flags = URB_ISO_ASAP;
        urb->transfer_buffer = ov->sbuf[n].data;
        urb->complete = ov51x_isoc_irq;
        urb->number_of_packets = FRAMES_PER_DESC;
        urb->transfer_buffer_length = ov->packet_size * FRAMES_PER_DESC;
        urb->interval = 1;
        for (fx = 0; fx < FRAMES_PER_DESC; fx++) {
            urb->iso_frame_desc[fx].offset = ov->packet_size * fx;
            urb->iso_frame_desc[fx].length = ov->packet_size;
        }
    }
    ov->streaming = 1;
    //提交 urb
    for (n = 0; n < OV511_NUMSBUF; n++) {
        ov->sbuf[n].urb->dev = ov->dev;
        err = usb_submit_urb(ov->sbuf[n].urb, GFP_KERNEL);
        if (err) {
            err("init isoc: usb_submit_urb(%d) ret %d", n, err);
            return err;
        }
    }
    return 0;
}
```

然后提交 urb 后，只要等待中断。

```
static void ov51x_isoc_irq(struct urb *urb, struct pt_regs *regs)
{
    int i;
    struct usb_ov511 *ov;
    struct ov511_sbuf *sbuf;
    if (!urb->context) {
        PDEBUG(4, "no context");
        return;
```

```
    }
    sbuf = urb->context;
    ov = sbuf->ov;
    if (!ov || !ov->dev || !ov->user) {
        PDEBUG(4, "no device, or not open");
        return;
    }
    if (!ov->streaming) {
        PDEBUG(4, "hmmm... not streaming, but got interrupt");
        return;
    }
   //检查 urb 的状态
   if (urb->status == -ENOENT || urb->status == -ECONNRESET) {
          PDEBUG(4, "URB unlinked");
          return;
   }
    if (urb->status != -EINPROGRESS && urb->status != 0) {
        err("ERROR: urb->status=%d: %s", urb->status,
           symbolic(urb errlist, urb->status));
    }
    //获取数据，将数据复制到帧缓存中
    PDEBUG(5, "sbuf[%d]: Moving %d packets", sbuf->n,urb->number of packets);
    for (i = 0; i < urb->number of packets; i++) {
        //如果当前帧号大于 0，则复制，否则不复制
        if (ov->curframe >= 0) {
            int n = urb->iso_frame_desc[i].actual_length;
            int st = urb->iso_frame_desc[i].status;
            unsigned char *cdata;
            urb->iso_frame_desc[i].actual_length = 0;
            urb->iso_frame_desc[i].status = 0;
            //计算地址
            cdata = urb->transfer buffer+ urb->iso frame desc[i].offset;
            if (!n) {
                PDEBUG(4, "Zero-length packet");
                continue;
            }
            if (st)PDEBUG(2, "data error: [%d] len=%d, status=%d",i, n, st);
            if (ov->bclass == BCL_OV511)
                ov511_move_data(ov, cdata, n);//复制数据到当前帧号的帧缓冲
            else if (ov->bclass == BCL_OV518)
                ov518_move_data(ov, cdata, n);
            else
                err("Unknown bridge device (%d)", ov->bridge);
        }
        else if (waitqueue_active(&ov->wq)) //唤醒等待
```

```
            {
                wake_up_interruptible(&ov->wq);
            }
        }
        //重新提交 urb，标志更改为 GFP_ATOMIC
        urb->dev = ov->dev;
        if ((i = usb_submit_urb(urb, GFP_ATOMIC)) != 0)
            err("usb submit urb() ret %d", i);
        return;
}
```

通过发送 VIDIOCSYNC 或 VIDIOCMCAPTURE 命令应用程序采集图像。

```
int ov51x_v4l1_ioctl_internal(struct inode *inode, struct file *file,unsigned int
cmd, void *arg)
{
    switch (cmd) {
        case VIDIOCMCAPTURE:
        ...
        return ov51x new frame(ov, f);
        case VIDIOCSYNC://阻塞直到出错或捕捉成功
        {
        unsigned int fnum = *((unsigned int *) arg);
        struct ov511_frame *frame;
        int rc;
        if (fnum >= OV511_NUMFRAMES) {
            err("VIDIOCSYNC: invalid frame (%d)", fnum);
            return -EINVAL;
        }
        //指向当前要采集的帧号
        frame = &ov->frame[fnum];
        PDEBUG(4, "syncing to frame %d, grabstate = %d", fnum,frame->grabstate);
        switch (frame->grabstate) {
        case FRAME UNUSED:
            return -EINVAL;
        case FRAME_READY:
        case FRAME_GRABBING:
        case FRAME_ERROR:
redo:
            if (!ov->dev)return -EIO;
            rc = wait_event_interruptible(frame->wq,
                (frame->grabstate == FRAME DONE)
                || (frame->grabstate == FRAME ERROR));
            if (rc)return rc;
            if (frame->grabstate == FRAME_ERROR) {//发生错误，捕捉下帧
                if ((rc = ov51x_new_frame(ov, fnum)) < 0)
```

```
                        return rc;
                    goto redo;
                }
            case FRAME_DONE://帧完成
                if (ov->snap_enabled && !frame->snapshot) {
                    if ((rc = ov51x_new_frame(ov, fnum)) < 0)
                        return rc;
                    goto redo;
                }
                frame->grabstate = FRAME_UNUSED;
                if ((ov->snap_enabled) && (frame->snapshot)) {
                    frame->snapshot = 0;
                    ov51x_clear_snapshot(ov);
                }
                //后处理
                ov51x_postprocess(ov, frame);
                break;
            } // 结束转换
            return 0;//图像捕捉成功
        }
}
```

VIDIOCMCAPTURE 命令的处理过程主要是调用了 ov51x_new_frame 来更新帧号与当前帧的状态。

```
static int  ov51x_new_frame(struct usb_ov511 *ov, int framenum)
{
    struct ov511_frame *frame;
    int newnum;
    PDEBUG(4, "ov->curframe = %d, framenum = %d", ov->curframe, framenum);
    if (!ov->dev)
        return -1;
    //如果另一帧数据已经准备就绪（ov->curframe! =-1），那就使用它代替
    if (ov->curframe == -1) {
        newnum = (framenum - 1 + OV511_NUMFRAMES) % OV511_NUMFRAMES;
        if (ov->frame[newnum].grabstate == FRAME_READY)
            framenum = newnum;
    } else
        return 0;
    //根据帧号得到图像数据
    frame = &ov->frame[framenum];
    PDEBUG(4, "framenum = %d, width = %d, height = %d", framenum,
          frame->width, frame->height);
    //设置为正在捕捉图像状态，避免多次进入
    frame->grabstate = FRAME_GRABBING;
```

```
        frame->scanstate = STATE SCANNING;
        frame->snapshot = 0;
        //更改当前帧号
        ov->curframe = framenum;
        if (frame->width > ov->maxwidth)
            frame->width = ov->maxwidth;
        frame->width &= ~7L;
        if (frame->height > ov->maxheight)
            frame->height = ov->maxheight;
        frame->height &= ~3L;
        return 0;
}
```

在应用程序中可以这样编程：

```
While(1)
{
    //发起捕捉
    ioctl(vd->fd, VIDIOCMCAPTURE, &(vd->mmap)) ;
    //调用 VIDIOCSYNC 等待一帧截取结束
    if(ioctl(vd->fd, VIDIOCSYNC, &frame) < 0)
     {
     perror("v4l sync:VIDIOCSYNC");
     return -1;
     }
     //处理 mmap 映射后的缓冲
}
```

编译方法很简单，只需要在编译内核时在 USB 设备类中选中 USB OV511 Camera support 即可，如图 5.8 所示。

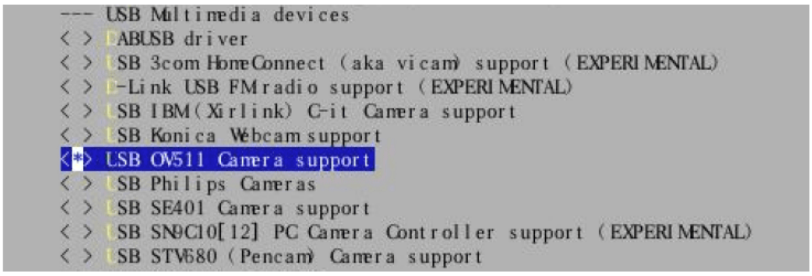


图 5.8　编译 OV511 驱动

5.6.4　spca5xx 编译与使用

spca5xx 是 Michel Xhaard 主持的一个开源的 Linux 下的摄像头驱动项目，该项目的主页是 http://mxhaard.free.fr。spca5xx 可以支持几百种摄像头，包括中星微的 ZC0303 芯片。spca5xx 还为

摄像头提供了一个完整的网络应用程序。下面介绍如何在嵌入式系统上使用这个项目。

下 载 http://mxhaard.free.fr/spca50x/embedded/KernelPatch/usb-2.4.31LE06.patch.tar.gz， 将 它解压：

```
# tar -xvzf usb-2.6.8.1-2.patch
```

接着打补丁，这里不将这个驱动编译进内核，而是编译成可加载模块的方式。

```
# patch -p1 < usb-2.6.8.1-2.patch
```

在压缩后的文件夹中增加 makefile 文件：

```
AR = ar
ARCH = arm
CC = arm-linux-gcc
obj-m := spca.o
spca-objs :=spca core.o spcadecoder.o
else
KERNELDIR ?= /home/s3c2410/linux-2.6.8.1
PWD := $(shell pwd)
modules:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) LDDINC=$(PWD)/../include modules
endif
```

执行：

```
#make
#make install
```

将 spca.ko 复制到目标板。

```
#insmod spca.ko
#mknod /dev/video0 c 81 0
```

下载 http://mxhaard.free.fr/spca50x/embedded/Servfox/ servfox-R1_1_3.tar.gz。修改 makefile，编译成功后，发送到目标板，执行：

```
servfox  -d /dev/video0  -s 320x240 -p 1 -o test.avi
```

这样就可以将采集的图像保存到 test.avi 中了。

5.7 USB Gadget

所谓 Gadget 驱动是指基于 Linux 的外围设备中的 USB 驱动代码。Linux 中的 USB Gadget API <linux/usb_gadget.h>使得外设和其他嵌入式系统作为从设备，运行变得容易。许多 Linux 系统无

法使用这些 API，因为它们只有 USB 主机控制器。但是如果平台是嵌入式系统，一个 USB 外设控制器通常是内嵌进 CPU 的。在这种情况下，USB Gadget API 将是最适合的选项。USB Gadget API 框架包含三个层次。

（1）外围控制器驱动：这个部分直接与硬件打交道，是与平台相关的层次不同的控制器硬件。其驱动不同，所以也称作 USB 设备控制器驱动。

（2）Gadget 驱动：这个层次通常是与硬件无关的。它调用外围控制器驱动，一个 Gadget 驱动实现一种或多种功能，比如作为网络连接或扬声器。

（3）上层：包括网络、文件系统、块设备 I/O 子系统。这些部分处理 Gadget 驱动传来的数据。

Linux 内核中包含一些公共的 Gadget 驱动，每个 Gadget 驱动都实现了一种单一的通用的 USB 功能，它们事实上可以用于任何 USB 外设控制器。这些驱动包括 Gadget Zero（控制器测试驱动）、Ethernet over USB（USB 网络设备）、Gadget FS（Gadget 文件系统）、File-backed Storage（USB 存储设备）、Serial（串行设备）、MIDI（MIDI 音乐播放）。Gadget 驱动使用如下的结构描述。

```
struct usb gadget driver {
char *function;//功能描述
enum usb_device_speed   speed;//驱动支持的最高速率
int (*bind)(struct usb_gadget *);
void (*unbind)(struct usb_gadget *);
int (*setup)(struct usb_gadget *,const struct usb_ctrlrequest *);
void (*disconnect)(struct usb_gadget *);
void (*suspend)(struct usb_gadget *);
void (*resume)(struct usb_gadget *);
struct device driver   driver;
};
```

bind 函数在驱动与一个 Gadget 绑定时激发，这时端点 0 已经初始化完毕。unbind 则与 bind 相反。setup 在处理非硬件驱动的端点 0 控制请求时激发，大多数的请求必须由 Gadget 层进行处理，比如描述符和配置管理。可以使用下面的函数注册与注销 Gadget 驱动。

```
int usb_gadget_register_driver (struct usb_gadget_driver *driver);
int usb_gadget_unregister_driver (struct usb_gadget_driver *driver);
```

可以看出 usb_gadget_driver 中的成员函数都包含 struct usb_gadget 参数。

```
struct usb gadget {
const struct usb_gadget_ops *ops;//操作函数集合
struct usb_ep   *ep0;//端点 0
struct list_head    ep_list;//端点列表
enum usb_device_speed       speed;//设备速度
unsigned    is_dualspeed:1;//同时支持高速和全速则为 1
unsigned    is_otg:1;//支持 OTG 为 1
unsigned    is_a_peripheral:1;//为 0，除非 is_otg=1
unsigned    b_hnp_enable:1;//HNP 使能的 A-Host
unsigned    a_hnp_support:1;//支持 HNP 的 A-host
```

```
unsigned    a alt hnp support:1;
const char  *name;
struct device   dev;
};
```

操作函数集合包括如下函数。

```
struct usb gadget ops {
int (*get_frame)(struct usb_gadget *);
int (*wakeup)(struct usb_gadget *);
int (*set_selfpowered) (struct usb_gadget *, int is_selfpowered);
int (*vbus_session) (struct usb_gadget *, int is_active);
int (*vbus_draw) (struct usb_gadget *, unsigned mA);
int (*pullup) (struct usb_gadget *, int is_on);
int (*ioctl)(struct usb_gadget *,unsigned code, unsigned long param);
};
```

Gadget API 的 USB 端点用下面的结构描述。

```
struct usb ep {
    void            *driver_data;
    const char      *name;//名称
    const struct usb_ep_ops *ops;//操作函数集合
    struct list_head    ep_list;//端点列表
    unsigned        maxpacket:16;//最大包
};
struct usb_ep_ops {
    int (*enable) (struct usb_ep *ep,const struct usb_endpoint_descriptor *desc);
    int (*disable) (struct usb_ep *ep);
    struct usb_request *(*alloc_request) (struct usb_ep *ep,gfp_t gfp_flags);
    void (*free_request) (struct usb_ep *ep, struct usb_request *req);
    void *(*alloc_buffer) (struct usb_ep *ep, unsigned bytes,dma_addr_t *dma, gfp_t
    gfp flags);
    void (*free_buffer) (struct usb_ep *ep, void *buf, dma_addr_t dma,unsigned
    bytes);
    int (*queue) (struct usb_ep *ep, struct usb_request *req,gfp_t gfp_flags);
    int (*dequeue) (struct usb_ep *ep, struct usb_request *req);
    int (*set_halt) (struct usb_ep *ep, int value);
    int (*fifo_status) (struct usb_ep *ep);
    void (*fifo_flush) (struct usb_ep *ep);
};//端点操作函数集合
```

5.7.1 USB 设备控制器驱动

可以通过分析 pxa2xx_udc.c 来掌握 USB 外围控制器驱动的开发方法。它是 PXA25X 处理器

的 USB 外围控制器驱动，它支持 16 个端点。首先要实现前面提到的 usb_gadget_register_driver 和 usb_gadget_unregister_driver。

```
int usb_gadget_register_driver(struct usb_gadget_driver *driver)
{
    struct pxa2xx_udc   *dev = the_controller;
    int retval;
    //检查 bind、setup 函数是否定义，设备支持的速度是否设置
    if (!driver|| driver->speed < USB SPEED FULL|| !driver->bind|| !driver->
    disconnect || !driver->setup)
        return -EINVAL;
    if (!dev)return -ENODEV;
    if (dev->driver)return -EBUSY;
    //设置 Gadget 的驱动
    dev->driver = driver;
    dev->gadget.dev.driver = &driver->driver;
    dev->pullup = 1;
    //向内核添加设备
    device_add (&dev->gadget.dev);
    retval = driver->bind(&dev->gadget);//绑定 Gadget
    if (retval) {
        DMSG("bind to driver %s -> error %d\n",
                driver->driver.name, retval);
        device_del (&dev->gadget.dev);
        dev->driver = NULL;
        dev->gadget.dev.driver = NULL;
        return retval;
    }
    device_create_file(dev->dev, &dev_attr_function);//创建设备文件
    DMSG("registered gadget driver '%s'\n", driver->driver.name);
    //现在可以允许主机在端点 0 的探测
    pullup(dev, 1);
    dump state(dev);
    return 0;
}
```

其次，注册一个 USB 设备控制器驱动。

```
static struct platform driver udc driver = {
    .probe       = pxa2xx udc probe,
    .shutdown    = pxa2xx udc shutdown,
    .remove      =  exit p(pxa2xx udc remove),
    .suspend     = pxa2xx udc suspend,
    .resume      = pxa2xx udc resume,
    .driver      = {
```

```
            .owner   = THIS MODULE,
            .name    = "pxa2xx-udc",
        },
    };
```

下面来看看 pxa2xx_udc_probe 的处理过程。它的主要作用是识别设备，并初始化 GPIO，最后申请中断。

```
static int _init pxa2xx_udc_probe(struct platform_device *pdev)
{
    struct pxa2xx_udc *dev = &memory;
    int retval, out_dma = 1, vbus_irq;
    u32 chiprev;
    //检查是否 XSCALE
    asm("mrc%? p15, 0, %0, c0, c0" : "=r" (chiprev));
    if ((chiprev & CP15R0 VENDOR MASK) != CP15R0 XSCALE VALUE) {
        printk(KERN ERR "%s: not XScale!\n", driver name);
        return -ENODEV;
    }
//检查产品版本
switch (chiprev & CP15R0_PRODREV_MASK) {
#if defined(CONFIG_ARCH_PXA)
    case PXA255_A0:
        dev->has_cfr = 1;
        break;
    case PXA250 A0:
    case PXA250 A1:
    case PXA250 B2: case PXA210 B2:
    case PXA250 B1: case PXA210 B1:
    case PXA250 B0: case PXA210 B0:
        out_dma = 0;
    case PXA250_C0: case PXA210_C0:
        break;
#elif   defined(CONFIG_ARCH_IXP4XX)
    case IXP425_A0:
    case IXP425_B0:
    case IXP465 AD:
        dev->has cfr = 1;
        out dma = 0;
        break;
#endif
    default:
        out_dma = 0;
        printk(KERN_ERR "%s: unrecognized processor: %08x\n",driver_name,
        chiprev);
```

```
        return -ENODEV;
    }
    dev->dev = &pdev->dev;
    dev->mach = pdev->dev.platform_data;
    //设置相关的 GPIO
    if (dev->mach->gpio_vbus) {
        vbus_irq = IRQ_GPIO(dev->mach->gpio_vbus & GPIO_MD_MASK_NR);
        pxa_gpio_mode((dev->mach->gpio_vbus & GPIO_MD_MASK_NR)| GPIO_IN);
        set_irq_type(vbus_irq, IRQT_BOTHEDGE);
    } else
        vbus_irq = 0;
    if (dev->mach->gpio_pullup)
        pxa_gpio_mode((dev->mach->gpio_pullup & GPIO_MD_MASK_NR)
                | GPIO_OUT | GPIO_DFLT_LOW);
    //初始化 UDC 的看门狗定时器
    init_timer(&dev->timer);
    dev->timer.function = udc_watchdog;
    dev->timer.data = (unsigned long) dev;
    //设备初始化
    device_initialize(&dev->gadget.dev);
    dev->gadget.dev.parent = &pdev->dev;
    dev->gadget.dev.dma_mask = pdev->dev.dma_mask;
    the_controller = dev;
    platform_set_drvdata(pdev, dev);
    udc_disable(dev);
    udc_reinit(dev);
    dev->vbus = is_vbus_present();
    //申请 USB 中断
    retval = request_irq(IRQ_USB, pxa2xx_udc_irq,IRQF_DISABLED, driver_name,
    dev);
    if (retval != 0) {
        printk(KERN_ERR "%s: can't get irq %i, err %d\n",driver_name, IRQ_USB,
        retval);
        return -EBUSY;
    }
    dev->got_irq = 1;
    if (vbus_irq) {
        retval = request_irq(vbus_irq, udc_vbus_irq,
                SA_INTERRUPT | SA_SAMPLE_RANDOM,driver_name, dev);
        if (retval != 0) {
            printk(KERN_ERR "%s: can't get irq %i, err %d\n",
                driver_name, vbus_irq, retval);
            free_irq(IRQ_USB, dev);
            return -EBUSY;
        }
```

```
        }
        //创建/proc 节点
        create_proc_files();
        return 0;
    }
```

struct pxa2xx_udc 展示了 pxa2xx 中的 USB 设备控制器的全部特性，也揭示了 USB 设备控制器驱动设计的秘密，pxa2xx_udc.c 就是围绕这个结构设计的。

```
struct pxa2xx_udc {
    struct usb_gadget           gadget;
    struct usb_gadget_driver    *driver;
    enum ep0_state      ep0state;
    struct udc stats    stats;
    unsigned     got irq : 1,vbus : 1,pullup : 1,has cfr : 1,
                 req pending : 1,req std : 1,req config : 1;
    struct timer list   timer;
    struct device       *dev;
    struct pxa2xx_udc_mach_info*mach;
    u64 dma_mask;
    struct pxa2xx_ep    ep [PXA_UDC_NUM_ENDPOINTS];
};
```

5.7.2 Gadget 驱动

本节介绍 USB Gadget Serial Driver，它是一个典型的 USB Gadget 层驱动。实际上它是 USB Gadget 层驱动和 TTY 驱动的结合体，也是一种 USB 转串口驱动，当这个 USB 设备挂在 USB 主机上，它可以被识别为一种串口终端。

```
static const struct tty_operations gs_tty_ops = {//TTY 操作集合
    .open =             gs_open,
    .close =            gs_close,
    .write =            gs_write,
    .put_char =         gs_put_char,
    .flush_chars =      gs_flush_chars,
    .write room =       gs write room,
    .ioctl =            gs ioctl,
    .set termios =      gs set termios,
    .throttle =         gs throttle,
    .unthrottle =       gs unthrottle,
    .break_ctl =        gs_break,
    .chars_in_buffer =  gs_chars_in_buffer,
};
static struct tty_driver *gs_tty_driver;
```

```
static struct usb_gadget_driver gs_gadget_driver = {//USB Gadget 驱动
#ifdef CONFIG_USB_GADGET_DUALSPEED
    .speed =        USB_SPEED_HIGH,
#else
    .speed =        USB_SPEED_FULL,
#endif /* CONFIG_USB_GADGET_DUALSPEED */
    .function =     GS_LONG_NAME,
    .bind =         gs_bind,
    .unbind =       gs_unbind,
    .setup =        gs_setup,
    .disconnect =   gs_disconnect,
    .driver = {
        .name =     GS_SHORT_NAME,
    },
};
```

在模块初始化的时候要注册两个设备驱动，一个为 Gadget Driver，一个为 TTY Driver。在 TTY 驱动完成它的操作的过程中调用 Gadget Driver 提供的接口，作为数据传输通道。

```
static int __init gs_module_init(void)
{
    int i;
    int retval;
    retval = usb_gadget_register_driver(&gs_gadget_driver);//注册 Gadget 驱动
    if (retval) {
        printk(KERN_ERR "gs_module_init: cannot register gadget driver, ret=%d\n",
        retval);
        return retval;
    }
    gs_tty_driver = alloc_tty_driver(GS_NUM_PORTS);
    if (!gs_tty_driver)
        return -ENOMEM;
    gs_tty_driver->owner = THIS_MODULE;
    gs_tty_driver->driver_name = GS_SHORT_NAME;
    gs_tty_driver->name = "ttygs";
    gs_tty_driver->major = GS_MAJOR;
    gs_tty_driver->minor_start = GS_MINOR_START;
    gs_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;//设备主类型
    gs_tty_driver->subtype = SERIAL_TYPE_NORMAL;//子类型
    //设置支持的 TTY 模式
    gs_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV;
    gs_tty_driver->init_termios = tty_std_termios;
    //设置终端的初始化参数，包括波特率、数据位等
    gs_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
    tty_set_operations(gs_tty_driver, &gs_tty_ops);
```

```
    for (i=0; i < GS NUM PORTS; i++)
        sema_init(&gs_open_close_sem[i], 1);
    retval = tty_register_driver(gs_tty_driver);//注册 TTY 驱动
    if (retval) {
        usb_gadget_unregister_driver(&gs_gadget_driver);
        put_tty_driver(gs_tty_driver);
        printk(KERN_ERR "gs_module_init: cannot register tty driver, ret=%d\n",
        retval);
        return retval;
    }
    return 0;
}
```

最后说明一下，在 USB Gadget 设备中各种 USB 描述符是由 Gadget Driver 层来管理的。比如设备描述符 gs_device_desc。

```
static struct usb device descriptor gs device desc = {
    .bLength =            USB DT DEVICE SIZE,
    .bDescriptorType =    USB_DT_DEVICE,
    .bcdUSB =             _constant_cpu_to_le16(0x0200),
    .bDeviceSubClass =    0,
    .bDeviceProtocol =    0,
    .idVendor =           _constant_cpu_to_le16(GS_VENDOR_ID),
    .idProduct =          _constant_cpu_to_le16(GS_PRODUCT_ID),
    .iManufacturer =      GS MANUFACTURER STR ID,
    .iProduct =           GS PRODUCT STR ID,
    .iSerialNumber =      GS SERIAL STR ID,
    .bNumConfigurations =    GS NUM CONFIGS,
};
```

驱动在 gs_bind 中设置描述符，例如：

```
gcnum = usb_gadget_controller_number(gadget);//获取 USB 设备控制器的 BCD 码
if (gcnum >= 0)
    gs_device_desc.bcdDevice =cpu_to_le16(GS_VERSION_NUM | gcnum);
else {
    printk(KERN WARNING "gs bind: controller '%s' not recognized\n",
    gadget->name);
    //没有识别，但通常是安全的
    gs device desc.bcdDevice = constant cpu to le16(GS VERSION NUM|0x0099);
}
usb_ep_autoconfig_reset(gadget);//复位配置
//选择一个适合该描述符的端点
ep = usb_ep_autoconfig(gadget, &gs_fullspeed_in_desc);
if (!ep)goto autoconf_fail;
EP_IN_NAME = ep->name;
```

```
ep->driver_data = ep;
ep = usb_ep_autoconfig(gadget, &gs_fullspeed_out_desc);
if (!ep)goto autoconf_fail;
EP_OUT_NAME = ep->name;
ep->driver_data = ep;
if (use_acm) {
    ep = usb_ep_autoconfig(gadget, &gs_fullspeed_notify_desc);
    if (!ep) {
        printk(KERN_ERR "gs_bind: cannot run ACM on %s\n", gadget->name);
        goto autoconf_fail;
    }
    gs_device_desc.idProduct = __constant_cpu_to_le16(GS_CDC_PRODUCT_ID),
    EP_NOTIFY_NAME = ep->name;
    ep->driver_data = ep;
}
//设备类别设置，厂商自定义设备，还是第二类设备
gs_device_desc.bDeviceClass = use_acm
    ? USB_CLASS_COMM : USB_CLASS_VENDOR_SPEC;
gs_device_desc.bMaxPacketSize0 = gadget->ep0->maxpacket;
```

第6章 Linux Framebuffer 驱动

在 Linux 内核中，Framebuffer 驱动是显示驱动的标准。Framebuffer 将显示设备抽象为帧缓冲区，用户通过内存映射将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反应在屏幕上。目前大多数的界面系统都支持 Framebuffer 驱动。本章重点介绍 Framebuffer 驱动的框架与开发方法，以及界面系统的架构。

6.1 LCD 原理

常见的液晶显示器按物理结构分为 4 种：（1）扭曲向列型（TN——Twisted Nematic）；（2）超扭曲向列型（STN——SuperTN）；（3）双层超扭曲向列型（DSTN——Dual Scan Tortuosity Nomograph）；（4）薄膜晶体管型（TFT——Thin Film Transistor）。前三种类型在名称上只有细微的差别，说明它们的显示原理具有很多共性。不同之处是液晶分子的扭曲角度各异。其中，DSTN 可以算是这三种类型的代表。由这种液晶体所构成的液晶显示器对比度和亮度仍比较差、可视角度较小、色彩也欠丰富，但它因结构简单、价格低廉，故还占有着一定市场。第 4 种 TFT 是现在最为常用的类型。TFT 是指液晶显示器上的每一液晶像素点都由集成在其后的薄膜晶体管来驱动。TFT 液晶显示器具有屏幕反应速度快、对比度好、亮度高、可视角度大、色彩丰富等特点，并克服了 DSTN 液晶显示器固有的一些弱点，比其他三种类型更具优势，是当前液晶显示器的主流设备。总地来说，STN 结构相对简单，生产成本低，只能显示一定的颜色深度，图像比较暗，对比度低，视角小、反应速度慢，不适合显示高速视频画面，只适宜文字处理和静态图像操作。而 TFT 是目前最好的 LCD，效果接近 CRT，可以适应视频监控等领域的应用。

STN 型 LCD 所使用单纯驱动电极的方式，都是采用 X、Y 轴的交叉方式来驱动的。它是靠占空比来显示灰度或颜色的，扫描时，每根数据线对应屏幕上的一个点，而这些点的颜色深度是依据相应数据线的占空比决定的。而 TFT 型 LCD 的每次像素时钟数据线上的数值对应一个点的颜色。

典型的 LCD 控制器都包含如下基本信号线。

（1）帧同步（FCLK）：表示显示帧的开始。

（2）行同步（LCLK）：表示行的开始。

（3）时钟（PCLK）：像素时钟。

（4）数据线（LDD[X:0]）：点的颜色数据。

6.2 Linux 下 LCD 驱动架构

Linux 提供了专门的 LCD 驱动类，即 Framebuffer 设备驱动程序，它被集中放置在/linux/drivers/video 目录下。Framebuffer 设备驱动提供给用户一个直接的面向显示缓冲区的统一接口。另外 linux/drivers/video/fbmem.c 文件提供了 LCD 驱动的通用文件操作接口，驱动程序可以实现自己的文件操作接口，也可使用 fbmem.c 中提供默认的接口。

```
//linux/drivers/video/fbmem.c
static struct file_operations fb_fops = {
    .owner= THIS_MODULE,
    .read = fb_read,
    .write= fb_write,
    .ioctl= fb_ioctl,
    .mmap = fb_mmap,
    .open = fb_open,
    .release = fb_release,
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
    .get_unmapped_area = get_fb_unmapped_area,
#endif
};
```

可以使用下列函数注册、卸载一个 Framebuffer 驱动程序：

```
int register_framebuffer(struct fb_info *fb_info);
int unregister_framebuffer(struct fb_info *fb_info);
```

在 Linux 驱动代码下有一个 skeletonfb.c 文件，它演示了开发 Framebuffer 设备驱动程序的框架，下面来分析下这个文件。

Framebuffer 驱动程序一般要定义一个 fb_fix_screeninfo 结构的变量，它代表输出设备自身的特性，包含识别符、缓存地址、显示类型、显示的颜色属性、加速标志等。

```
static struct fb_fix_screeninfo xxxfb_fix __initdata = {
    .id = "FB's name", //识别符
    .type = FB_TYPE_PACKED_PIXELS,//显示类型
    .visual = FB_VISUAL_PSEUDOCOLOR,//显示的颜色属性
    .xpanstep = 1,
    .ypanstep = 1,
    .ywrapstep= 1,
    .accel = FB_ACCEL_NONE,//加速选项
};
```

在程序中定义一个 struct xxx_par;用来保存图形卡的硬件状态信息，常包含寄存器信息。然后定义一个 xxx_par 型的变量。

```
static struct xxx_par __initdata current_par;
```

现代图形卡不仅支持单通道，也支持多显示器，每个显示器拥有独立的数据区。这种情况下，每个显示器可用一个 Framebuffer 驱动和一个单独的 fb_info 结构描述。如果显卡支持多屏幕，则需要定义一个 fb_info 数组。

```
struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var;          //当前 var
    struct fb_fix_screeninfo fix;          //当前 fix
    struct fb_monspecs monspecs;           //当前 Monitor specs
    struct fb_cursor cursor;               //当前光标
    struct work_struct queue;              //Framebuffer 事件队列
    struct timer_list cursor_timer;        //光标定时器
    struct fb_pixmap pixmap;               //图像硬件调色板
    struct fb_pixmap sprite;               //光标硬件调色板
    struct fb_cmap cmap;
    struct list head modelist;
    struct fb_ops *fbops;
    char _iomem *screen_base;              //虚拟地址
    unsigned long screen_size;             //映射尺寸
    int currcon;
    void *pseudo_palette;                  //16 色模式下的假调色板
#define FBINFO STATE RUNNING    0
#define FBINFO STATE SUSPENDED  1
    u32 state;                             //硬件状态
    void *par;
};
```

这里定义一个全局的 struct fb_info，这个结构是 Framebuffer 驱动的核心。

```
static struct fb_info info;
```

Framebuffer 驱动有自己特有的文件操作接口 fb_ops：

```
static struct fb_ops xxxfb_ops = {
    .owner = THIS_MODULE,
    .fb_open = xxxfb_open,
    .fb_read = xxxfb_read,
    .fb write = xxxfb write,
    .fb_release = xxxfb_release,
    .fb_check_var = xxxfb_check_var,
    .fb_set_par = xxxfb_set_par,
    .fb_setcolreg = xxxfb_setcolreg,
```

```
        .fb_blank       = xxxfb_blank,
        .fb_pan_display = xxxfb_pan_display,
        //以下加粗的是必须实现的函数
        .fb_fillrect  = xxxfb_fillrect,
        .fb_copyarea  = xxxfb_copyarea,
        .fb_imageblit = xxxfb_imageblit,
        .fb_cursor    = xxxfb_cursor,
        .fb_rotate    = xxxfb_rotate,
        .fb_poll      = xxxfb_poll,
        .fb_sync      = xxxfb_sync,
        .fb_ioctl     = xxxfb_ioctl,
        .fb_mmap      = xxxfb_mmap,
    };
```

当 Framebuffer 驱动初次访问时调用 xxxfb_open。通常不必提供这个函数，一个使用它的情况是从文本模式切换到图形模式。

```
    static int xxxfb_open(const struct fb_info *info, int user)
```

当关闭/dev/fb 时调用 xxxfb_release。通常不必提供这个函数，一个使用它的情况是从图形模式切换到文本模式。

```
    static int xxxfb_release(const struct fb_info *info, int user)
```

下面的函数用来检查输入参数的正确性。它不改变硬件的状态。如果硬件不支持输入的模式，则返回-EINVAL。

```
    static int xxxfb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
    {
        const struct xxx_par *par = (const struct xxx_par *) info->par;
    }
```

xxxfb_set_par 用来更改硬件设置。它可以更改 xxx_par、fb_fix_screeninfo 结构中的信息，但是不能更改 fb_info 中的信息。这个函数一般在 xxxfb_check_var 之后使用。

```
    static int xxxfb_set_par(struct fb_info *info)
    {
        struct xxx_par *par = (struct xxx_par *) info->par;
    }
```

xxxfb_setcolreg 设置第 regno 个调色板寄存器。需要考虑的是驱动中有多少调色板寄存器。真彩色模式（FB_VISUAL_TRUECOLOR）不支持调色板。struct fb_info 的 pseudo_palette 成员表示 16 种颜色模式（控制台就是 16 种颜色模式）的假调色板。在假彩色（FB_VISUAL_PSEUDOCOLOR）和直接颜色模式(FB_VISUAL_DIRECTCOLOR)下，当前调色板存放在 fb_info 中的 struct fb_cmap cmap，可以对其定义每个像素值显示出的颜色。如果已经拥有一个静态的颜色映射表，则不需要实现这个函数。

```
static int xxxfb_setcolreg(unsigned regno, unsigned red, unsigned green,unsigned
blue, unsigned transp,const struct fb_info *info)
{
//超过范围
    if (regno >= 256)
       return 1;
   //可在此处做透明度处理
   //灰度模式显示
    if (info->var.grayscale) {
       /* grayscale = 0.30*R + 0.59*G + 0.11*B */
       red = green = blue = (red * 77 + green * 151 + blue * 28) >> 8;
    }
   //其他颜色类型
#define CNVT_TOHW(val,width) ((((val)<<(width))+0x7FFF-(val))>>16)
    switch (info->fix.visual) {
       case FB_VISUAL_TRUECOLOR:
       case FB_VISUAL_PSEUDOCOLOR:
               red = CNVT_TOHW(red, info->var.red.length);
               green = CNVT_TOHW(green, info->var.green.length);
               blue = CNVT_TOHW(blue, info->var.blue.length);
               transp = CNVT_TOHW(transp, info->var.transp.length);
               break;
       case FB_VISUAL_DIRECTCOLOR:
            // 此处范例表现为 8 位 DAC, 结果可能有所不同
       // for your hardware
               red = CNVT_TOHW(red, 8);
               green = CNVT_TOHW(green, 8);
               blue = CNVT_TOHW(blue, 8);
               //hey, there is bug in transp handling...
               transp = CNVT_TOHW(transp, 8);
               break;
    }
#undef CNVT_TOHW
    //真彩色的调色板与硬件无关
    if (info->fix.visual == FB_VISUAL_TRUECOLOR) {
       u32 v;
       if (regno >= 16)
           return 1;
       //将 RGB 颜色值编码成单值
       v = (red << info->var.red.offset) |
           (green << info->var.green.offset) |
           (blue << info->var.blue.offset) |
           (transp << info->var.transp.offset);
       switch (info->var.bits_per_pixel) {
         case 8:
                //一些手持设备支持此种模式
```

```
                ((u8*)(info->pseudo_palette))[regno] = v;
                break;
            case 16:
                ((u16*)(info->pseudo_palette))[regno] = v;
                break;
            case 24:
            case 32:
                ((u32*)(info->pseudo_palette))[regno] = v;
                break;
        }
        return 0;
    }
    return 0;
}
```

xxxfb_pan_display 根据 xoffset 和 yoffset 来隐藏或展开显示，依 vmode 参数而定。

```
static int xxxfb_pan_display(struct fb_var_screeninfo *var,const struct fb_info *info)
```

xxxfb_blank 函数功能：blank_mode！=0 则显示空白。否则反之。

```
static int xxxfb_blank(int blank_mode, const struct fb_info *info)
```

以下为加速函数，如果硬件不支持加速，可以用常规的操作代替。xxxfb_fillrect 用来进行区域填充。它根据 fb_fillrect 在屏幕上放置或移去一个矩形。

```
struct fb fillrect {
    _u32 dx;          //填充区域左上角的 X 坐标
    _u32 dy;          //填充区域左上角的 Y 坐标
    _u32 width;       //填充区域的宽
    _u32 height;      //填充区域的高
    _u32 color;       //填充颜色
    _u32 rop;         //光栅操作
};
void xxfb_fillrect(struct fb_info *p, const struct fb_fillrect *region)
```

xxxfb_copyarea 用来进行区域复制。

```
struct fb copyarea {
    _u32 dx;          //目标区域左上角的 X 坐标
    _u32 dy;          //目标区域左上角的 Y 坐标
    _u32 width;       //复制目标区域的宽
    _u32 height;      //复制目标区域的高
    _u32 sx;          //源区域的左上角的 X 坐标
    _u32 sy;          //源区域的左上角的 Y 坐标
};
void xxxfb_copyarea(struct fb_info *p, const struct fb_copyarea *area)
```

xxxfb_imageblit 在屏幕上画一幅图片。

```
struct fb_image {
    _u32 dx;                 //左上角的 X 坐标
    _u32 dy;                 //左上角的 Y 坐标
    _u32 width;              //图片的宽
    _u32 height;             //图片的高
    _u32 fg_color;           //用于单色位图
    _u32 bg_color;
    _u8  depth;              //像素深度
    const char *data;        //图像数据
    struct fb_cmap cmap;     //颜色映射信息
};
void xxxfb_imageblit(struct fb_info *p, const struct fb_image *image)
```

xxxfb_cursor 用来变更光标的属性。

```
struct fb_cursor {
    _u16 set;
    _u16 enable;             //打开/关闭光标
    _u16 rop;
    const char *mask;        //掩蔽码
    struct fbcurpos hot;
    struct fb_image image;   //光标图像结构
    char  *data;             //图像数据
};
int xxxfb_cursor(struct fb_info *info, struct fb_cursor *cursor)
```

xxxfb_rotate 用来支持屏幕旋转。

```
void xxxfb_rotate(struct fb_info *info, int angle)
```

xxxfb_poll 用于等待某个特殊的硬件事件。

```
void xxxfb_poll(struct fb_info *info, poll_table *wait)
```

通常加速引擎会占用一段时间，在向 Framebuffer 进行写操作之前，必须等待加速器完成它的操作。方式是使用 xxxfb_sync。

```
void xxxfb_sync(struct fb_info *info)
```

模块初始化：

```
int __init xxxfb_init(void)
{
   int cmap_len, retval;
   //通常使用 ioremap 将 LCD 控制器的物理地址映射为虚拟地址
   info.screen_base = framebuffer_virtual_memory;
```

```
    info.fbops = &xxxfb ops;
    info.fix = xxxfb_fix;
    info.pseudo_palette = pseudo_palette;
    //加速标志和模块标志
    info.flags = FBINFO_DEFAULT;
    info.par = current_par;
    //默认选项
    if (!mode option)
    mode option = "640x480@60";
    //寻找一个有效的视频模式
    retval = fb find mode(&info.var, &info, mode option, NULL, 0, NULL, 8);
    if (!retval || retval == 4)
    return -EINVAL;
    //分配映射表
    fb_alloc_cmap(&info.cmap, cmap_len, 0);
    info.var = xxxfb_var;
    //注册 framebuffer 驱动
    if (register_framebuffer(&info) < 0)
    return -EINVAL;
    printk(KERN INFO "fb%d: %s frame buffer device\n", info.node,
       info.fix.id);
    return 0;
}
```

模块卸载：

```
static void _exit xxxfb_cleanup(void)
{
      unregister_framebuffer(info);
}
```

6.3 S3C2410X LCD 控制器

S3C2410X LCD 控制器是一个将图像数据从系统内存传输到外部 LCD 驱动器的逻辑单元，它支持二值图像、4 级、16 级灰度图像和 256 色、4 096 色 STN LCD。也支持 1、2、4、8、16、24 位的 TFT LCD 屏。图 6.1 是 S3C2410X 的 LCD 控制器的原理图。

REGBANK 有 17 个可编程的寄存器和一个 256*16 的调色板。LCDCDMA 是一个专用的 DMA，可以自动将视频数据传输到 LCD 驱动器。VIDPRCS 接收 LCDCDMA 的视频数据，并将数据以合适的格式送到数据线 VD[23：0]。TIMEGEN 单元主要负责各种同步时序的生成。LCDCDMA 中包含了 FIFO 内存，当 FIFO 不满时，LCDCDMA 每次从帧缓存提取 16 字节的数据。FIFO 内存共 28 字（一字共 4 字节），包括 12 字 FIFOL 和 16 字 FIFOH。这两个 FIFO 是用来支持双扫描显示的，在单扫描情况下，只用到 FIFOH。

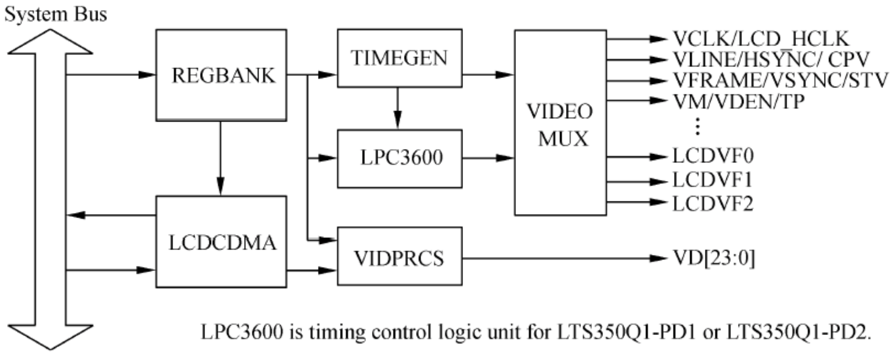


图 6.1　S3C2410X 的 LCD 控制器的原理图

下面重点介绍 TFT LCD 控制器操作。以 16BPP 模式为例子，这个模式下每个像素占两个字节。图 6.2 是 16BPP 模式下的屏幕上的点分布图。对应于视频缓冲，有两种存放方式，由 HWSWP 决定。图 6.3 为视频内存分布。图 6.4 显示了 16BPP 模式下的 LCD 数据线与颜色的对应关系。

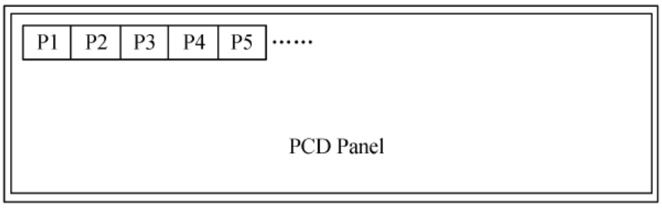


图 6.2　16BPP 模式下的屏幕像素分布

	D[31:16]	D[15:0]
000H	P1	P2
004H	P3	P4
008H	P5	P6
…		

BSWP＝0，HWSWP＝0

	D[31:16]	D[15:0]
000H	P2	P1
004H	P4	P3
008H	P6	P5
…		

BSWP＝0，HWSWP＝1

图 6.3　视频内存分布

VD	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RED	4	3	2	1	0		NC								NC								NC	
GREEN									5	4	3	2	1	0										
BLUE																	4	3	2	1	0			

图 6.4　16 位（5：6：5）下的管脚分布

S3C2410X 提供了电源使能功能（PWREN 引脚），当 LCD_PWREN 引脚连接在 LCD 显示器的电源控制脚，LCD 显示器的电源就由 S3C2410X 的 ENVID 自动控制。S3C2410X 还支持 LCDCON5 的 INVPWREN 位来调整 PWREN 信号的电平，如图 6.5 所示为 LCD 的电源控制图。

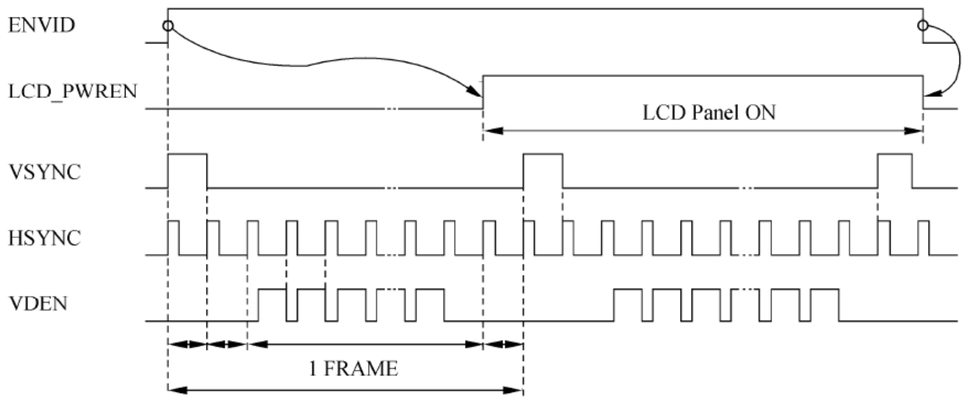


图 6.5　LCD 的电源控制

图 6.6 是典型的 S3C2410X 的 TFT 时序。S3C2410X 的 LCD 控制信号包括：

- VSYNC 为 LCD 控制器和 LCD 驱动器之间的帧同步信号。
- HSYNC 为 LCD 控制器和 LCD 驱动器之间的行同步脉冲信号。
- VCLK 代表 LCD 控制器和 LCD 驱动器之间的像素时钟信号，由 LCD 控制器送出的数据在 VCLK 的上升沿处送出，在 VCLK 的下降沿处被 LCD 驱动器采样。

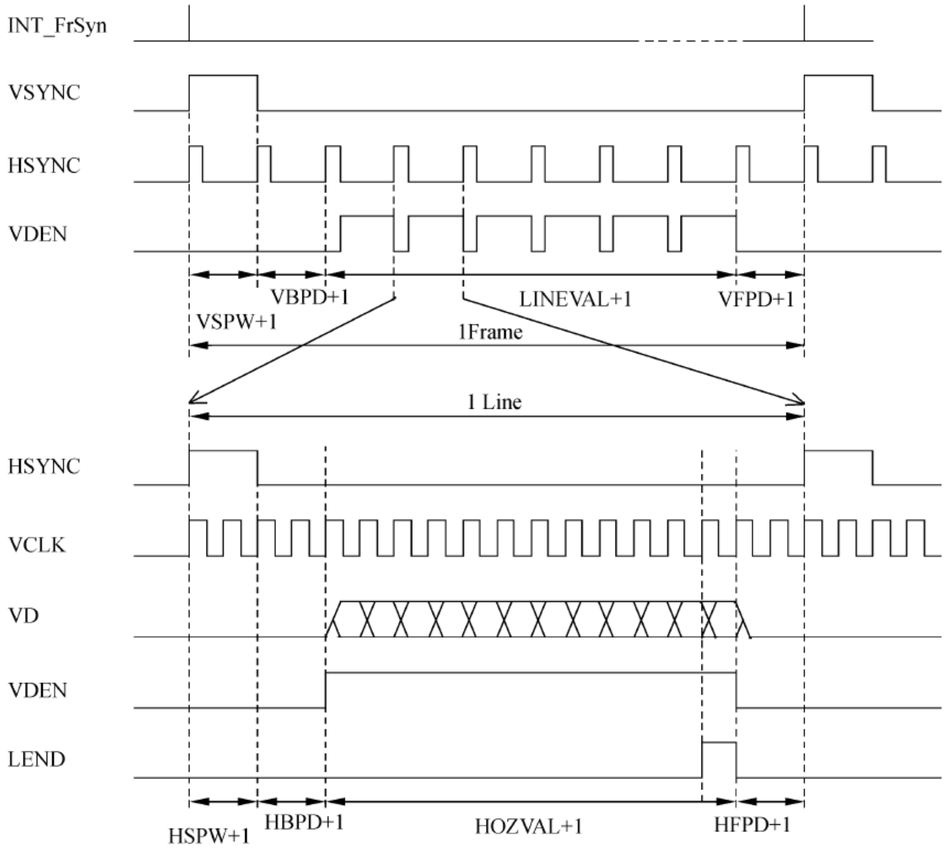


图 6.6　S3C2410X TFT 典型时序

- VDEN 则是 LCD 驱动器的 AC 信号，它可以被 LCD 驱动器用于改变行和列的电压极性，从而控制像素点的显示或熄灭。
- VD 为数据线，共 24 条。
- LEND 为行结束信号。

S3C2410X 的主要 LCD 控制寄存器包括 LCDCON1～LCDCON5，分别如表 6.1～6.5 所示。

1. LCD 控制寄存器 1（LCDCON1）

表 6.1　LCDCON1 描述

LCDCON1	位	描述	初始值
LINECNT（只读）	[27:18]	提供行计数器的状态，从 LINEVAL 到 0	0000000000
CLKVAL	[17:8]	决定 VCLK 和 HCLK 的换算。 STN：VCLK=HCLK/（CLKVAL×2） （CLKVAL>=2）； TFT：VCLK=HCLK/[（CLKVAL+1）×2] （CLKVAL>=0）	0000000000
MMODE	[7]	VM 信号的翻转频率（STN 型用）： 0=每帧； 1=由 MVAL 决定	0
PNRMODE	[6:5]	选择显示模式： 00=4 位双扫描 STN； 01=4 位单扫描 STN； 10=8 位单扫描 STN； 11=TFT 型	00
BPPMODE	[4:1]	选择 BPP（每像素的位数）： 0000=1 比特，STN，单色； 0001=2 比特，STN，4 级灰度； 0010=4 比特，STN，16 级灰度； 0011=8 比特，STN，彩色； 0100=12 比特，STN，彩色； 0000=1 比特，TFT； 0001=2 比特，TFT； 0010=4 比特，TFT； 0011=8 比特，TFT； 0100=16 比特，TFT； 0101=24 比特，TFT；	0000
ENVID	[0]	LCD 控制器允许： 0=禁止 LCD 信号输出； 1=允许 LCD 信号输出；	0

2. LCD 控制寄存器 2（LCDCON2）

表 6.2　　LCDCON2 描述

LCDCON2	位	描述	初始值
VBPD	[31:24]	TFT:帧同步后，帧数据开始前，无效行信号的数量； STN:必须为 0	0x00
LINEVAL	[23:14]	LCD 面板的垂直尺寸	0000000000
VFPD	[13:6]	TFT: 帧数据结束后，帧同步前，无效行信号的数量； STN:必须为 0	00000000
VSPW	[5:0]	TFT: 通过计算无效行的数量，决定帧同步信号脉冲高电平的宽度； STN:必须为 0	000000

3. LCD 控制寄存器 3（LCDCON3）

表 6.3　　LCDCON3 描述

LCDCON3	位	描述	初始值
HBPD（TFT） WDLY（STN）	[25:19]	TFT: 行同步下降沿后，行数据开始前，无效 VCLK 的数量 STN: WDLY[1：0]决定 VLINE 和 VCLK 之间的延迟。 00=16HCLK；01=32HCLK；10=64HCLK；11=128HCLK	0000000
HOZVAL	[18:8]	TFT/STN：LCD 面板的水平尺寸	00000000000
HFPD（TFT） LINEBLANK（STN）	[7:0]	TFT: 行数据结束后，行同步上升沿前，无效 VCLK 的数量 STN:表示每行内插入的空时间，这些位可以精确地调整 VLINE 的频率 BLANKTIME= LINEBLANK×HCLK×8	0x00

4. LCD 控制寄存器 4（LCDCON4）

表 6.4　　LCDCON4 描述

LCDCON4	位	描述	初始值
MVAL	[15:8]	STN: MMOD=1 时，决定 VM 信号的翻转频率	0x00
HSPW（TFT） WLH（STN）	[7:0]	TFT: 决定行同步信号脉冲高电平的宽度。单位是 VCLK STN:WLH[1:0]决定 VLINE 脉冲高电平的宽度，单位是 HCLK 00=16HCLK，01=32HCLK，10=64HCLK，11=128HCLK	0x00

5. LCD 控制寄存器 5（LCDCON5）

表 6.5　　LCDCON5 描述

LCDCON5	位	描述	初始值
保留	[31:17]	应该为 0	0
VSTATUS	[16:15]	TFT:帧状态。 00=VSYNC；01=BACK Porch； 10=有效；11=FRONT Porch；	00

续表

LCDCON5	位	描述	初始值
HSTATUS	[14:13]	TFT:行状态。 00=HSYNC；01=BACK Porch； 10=有效；11=FRONT Porch	00
BPP24BL	[12]	24bpp 视频内存的存放顺序 0=LSB；1=MSB	0
FRM565	[11]	TFT:选择 16bpp 时的输出格式： 0=5:5:5:1；1=5:6:6	0
INVVCLK	[10]	STN/TFT:控制 VCLK 的有效电平。 0=在 VCLK 的下降沿读数据； 1=在 VCLK 的上升沿读数据	0
INVVLINE	[9]	STN/TFT:VLINE/HSYNC 脉冲的电平。 0=正常；1=翻转	0
INVVFRAME	[8]	STN/TFT:VFRAME/VSYNC 脉冲的电平。 0=正常；1=翻转	0
INVVD	[7]	STN/TFT:数据线脉冲的电平。 0=正常；1=翻转	0
INVVDEN	[6]	TFT: VDEN 脉冲的电平。 0=正常；1=翻转	0
INVPWREN	[5]	STN/TFT:PWREN 脉冲的电平。 0=正常；1=翻转	0
INVLEND	[4]	TFT: LDEN 脉冲的电平。 0=正常；1=翻转	0
PWREN	[3]	STN/TFT:LCD_PWREN 信号输出允许。 0=禁止；1=允许	0
ENLEND	[2]	TFT:LEND 信号输出允许。 0=禁止；1=允许	0
BSWP	[1]	STN/TFT:Byte swap 控制位。 0=禁止；1=允许	0
HWSWP	[0]	STN/TFT:Half-Word swap 控制位。 0=禁止；1=允许	0

在设置 LCD 的显示参数时，注意 HOZVAL 和 LNEVAL 的值由 LCD 屏的尺寸决定，VCLK 信号的频率则是 HCLK 和 LCDCON1 寄存器中的 CLKVAL 域共同决定的。其中 CLKVAL 的最小值是 0。

```
HOZVAL=水平显示尺寸-1
LINEVAL=垂直显示尺寸-1
VCLK(Hz)=HCLK/[(CLKVAL+1)×2]
```

至于帧频率，其实就是 VSYNC 信号的频率，它与 LCDCON1 和 LCDCON2 / 3 / 4 寄存器的 VSYNC、VB2PD、VFPD、LINEVAL、HSYNC、HBPD、HFPD、HOZVAL 和 CLKVAL 都有关

系。大多数 LCD 驱动器都需要有与显示器相匹配的帧频率。S3C2410X 手册上给出的计算公式是：

```
Frame Rate = 1/ [ { (VSPW+1) + (VBPD+1) + (LIINEVAL + 1)+(VFPD+1)}.{(HSPW+1) +
(HBPD +1)+ (HFPD+1) + (HOZVAL + 1) }.{2.( CLKVAL+1 ) / ( HCLK ) } ]
```


6.4 S3C2410X LCD 驱动开发

现在在 linux/drivers/video/vfb.c 的基础上开发 S3C2410X 的 LCD 驱动。这里只说明更改的部分。首先定义一个 fb_var_screeninfo 结构和一个 fb_fix_screeninfo 结构：

```
static struct fb var screeninfo S3C2410fb default  initdata = {
    .xres =          240,
    .yres =          320,
    .xres_virtual = 240,
    .yres_virtual = 320,
    .bits_per_pixel =   16,
    .red =       { 11, 5, 0 },
    .green =     { 5, 6, 0 },
    .blue =      { 0, 5, 0 },
    .activate = FB ACTIVATE NOW,
    .height =   -1,
    .width =    -1,
    .pixclock = 20000,
    .left margin =  64,
    .right_margin = 64,
    .upper_margin = 32,
    .lower_margin = 32,
    .hsync_len =    64,
    .vsync_len =    2,
    .vmode = FB_VMODE_NONINTERLACED,
};
static struct fb fix screeninfo S3C2410fb fix  initdata = {
    .id =        "2410fb",
    .type =      FB TYPE PACKED PIXELS,
    .visual =    FB VISUAL TRUECOLOR,
    .xpanstep = 0,
    .ypanstep = 0,
    .ywrapstep= 0,
    .accel =     FB_ACCEL_NONE,
    .type_aux=  0,
};
```

在模块初始化的时候注册一个驱动类别：

```
static struct device driver S3C2410fb driver = {
    .name   = "S3C2410fb",
    .bus    = &platform bus type,
    .probe  = S3C2410fb probe,
    .remove = S3C2410fb_remove,
};
static struct platform_device S3C2410fb_device = {
    .name = "S3C2410fb",
    .id = 0,
    .dev = {
        .release = S3C2410fb platform release,
        .coherent dma mask=0xFFFFFFFF,
    }
};
ret = driver register(&S3C2410fb driver);
platform_device_register(&S3C2410fb_device);
```

设置 LCD 控制器前必须先禁止 LCD 控制器，设置工作结束后，再允许 LCD 控制器：

```
static void init_2410LCD(struct fb_info *info)
{
    #define LCDCON_1  ((7<<8)|(3<<5)|(12<<1))
    #define LCDCON_2  (2<<24)|(319<<14)|(2<<6)|4
    #define LCDCON 3  (8<<19)|(239<<8)|8
    #define LCDCON 4  (13<<8)|6
    #define LCDCON 5  (1<<11)|(0<<9)|(0<<8)|(0<<6)|(1)
    //先禁止控制器
     raw writel(LCDCON 1&(~S3C2410 LCDCON1 ENVID), S3C2410 LCDCON1);
     raw writel(lcdcon2, S3C2410 LCDCON2);
    _raw_writel(lcdcon3, S3C2410_LCDCON3);
    _raw_writel(LCDCON_4, S3C2410_LCDCON4);
    _raw_writel(LCDCON_5, S3C2410_LCDCON5);
    //允许控制器
    _raw_writel(LCDCON_1|S3C2410_LCDCON1_ENVID, S3C2410_LCDCON1);
}
static void S3C2410 setup gpio(void)
{
    DPRINTK("setup gpio\n");
     raw writel(0xaaaaaaaa, S3C2410 GPDCON);
    _raw_writel(3, S3C2410_LCDINTMSK);      //禁止 LCD 子中断
    _raw_writel(0, S3C2410_TPAL);           //禁用临时调色板
    _raw_writel(0, S3C2410_LPCSEL);     //禁止 LPC3600
}
```

对 LCD 文件操作接口如图 6.7 所示。其中 fb_fillrect、fb_copyarea、fb_imageblit、fb_cursor

几个必须实现的函数，采用 Linux 内核中通用的操作函数 cfb_fillrect、cfb_fillrect、cfb_imageblit、soft_cursor。

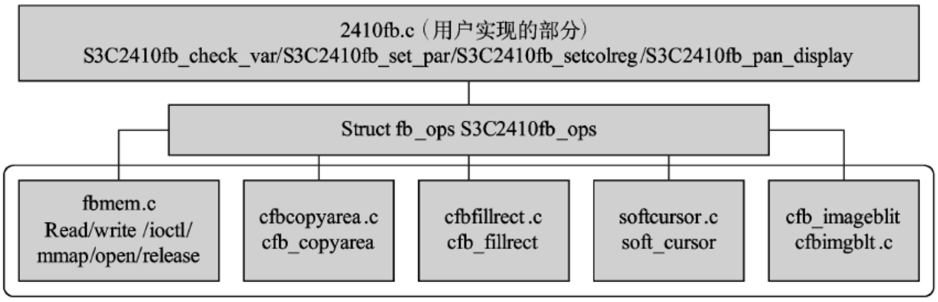


图 6.7 Framebuffer 驱动接口实现

```
static struct fb_ops S3C2410fb_ops = {
    .fb_check_var   = S3C2410fb_check_var,
    .fb_set_par     = S3C2410fb_set_par,
    .fb_setcolreg   = S3C2410fb_setcolreg,
    .fb_pan_display = S3C2410fb_pan_display,
    .fb_fillrect    = cfb_fillrect,
    .fb_copyarea    = cfb_copyarea,
    .fb_imageblit   = cfb_imageblit,
    .fb_cursor      = soft_cursor,
};
```

S3C2410fb_probe 探测函数主要完成 LCD 控制器的初始化、显示缓存的分配，最后注册一个 Framebuffer 驱动：

```
static int _init S3C2410fb_probe(struct device *device)
{
    struct platform_device *dev = to_platform_device(device);
    struct fb_info *info;
    int retval = -ENOMEM;
    //分配地址
    U32 pVideoBuffer;
    videomemorysize=PAGE_ALIGN(videomemorysize+PAGE_SIZE);
    videomemory=dma_alloc_writecombine(device,videomemorysize,&pVideoBuffer,
    GFP_KERNEL);
    if(videomemory==NULL)return retval;
    memset(videomemory, 0, videomemorysize);
    info = framebuffer_alloc(sizeof(u32) * 256, &dev->dev);
    if (!info)goto err;
    //填充 struct fb_info
    info->var = S3C2410fb_default;
    info->fix = S3C2410fb_fix;
    info->fbops = &S3C2410fb_ops;
    info->pseudo_palette = info->par;
    info->par = NULL;
```

```
    info->flags = FBINFO FLAG DEFAULT;
    //计算地址
    info->screen_base =videomemory+PAGE_SIZE;
    printk("videomemory=0x%08x, page_size=0x%08x, screen_base=0x%08x\n",
    videomemory,PAGE_SIZE,info->screen_base);
    info->fix.smem_start=pVideoBuffer+PAGE_SIZE;
    info->fix.smem_len=info->var.xres*info->var.yres*info->var.bits_per_pixel/8;
    printk("pVideoBuffer=0x%08x, smem len=0x%08x, smem start=0x%08x\n",
                pVideoBuffer,info->fix.smem_len,info->fix.smem_start);
    //初始化参数
    S3C2410fb_check_var(&info->var,info);
    S3C2410fb_set_par(info);
    S3C2410 setup gpio();
    init 2410LCD(info);
    retval = fb alloc cmap(&info->cmap, 256, 0);
    if (retval < 0)goto err1;
    retval = register_framebuffer(info);
    if (retval < 0)goto err2;
    dev_set_drvdata(&dev->dev, info);
    printk(KERN_INFO"fb%d: Virtual frame buffer device, using %ldK
          of video memory\n",info->node, videomemorysize >> 10);
    return 0;
err2:
    fb_dealloc_cmap(&info->cmap);
err1:
    framebuffer_release(info);
err:
    return retval;
}
```

卸载函数主要是完成注销设备、释放资源的功能：

```
static int S3C2410fb_remove(struct device *device)
{
    struct fb_info *info = dev_get_drvdata(device);
    if (info) {
        unregister framebuffer(info);
        vfree(videomemory);
        framebuffer release(info);
    }
    return 0;
}
```

S3C2410fb_set_par 用来更改硬件设置：

```
static int S3C2410fb_set_par(struct fb_info *info)
{
```

```
    info->fix.line length = get line length(info->var.xres virtual,info->var.
    bits_per_pixel); return 0;
}
```

测试程序如下：

```
int main(int argc, char *argv[])
{
    int i, fd, fbfd;
    struct fb var screeninfo vinfo;
    struct fb fix screeninfo finfo;
    _u8 *fb_buf;
    int fb_xres,fb_yres,fb_bpp;
    _u32 screensize;
    fbfd = open("/dev/fb1", O_RDWR);
    if (fbfd < 0) {
        fbfd = open("/dev/fb/0", O_RDWR);
        if(fbfd<0) {
            printf("Error: cannot open framebuffer device.\n");
                return -1;
        }
    }
    // 获取 fb_fix_screeninfo
    if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
        printf("Error reading fixed information.\n");
        close(fbfd);
            return -1;
    }
    // 获取 fb_var_screeninfo
    if (ioctl(fbfd, FBIOGET VSCREENINFO, &vinfo)) {
        printf("Error reading variable information.\n");
        close(fbfd);
        return -1;
    }
    printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits per pixel );
    fb_xres = vinfo.xres;
    fb_yres = vinfo.yres;
    fb_bpp  = vinfo.bits_per_pixel;
    //计算屏幕尺寸
    screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;
    fb_buf = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
                fbfd, 0);
    if ((int)fb buf == -1) {
        printf("Error: failed to map framebuffer device to memory.\n");
        close(fbfd);
            return -1;
```

```
    }
    memset(fb_buf,200,screensize);
    printf("ummap framebuffer device to memory.\n");
    sleep(10);
    munmap(fb_buf, screensize);
    close(fbfd);
    return 0;
}
```

测试结果是屏幕全部被绘成深红色。

6.5 基于 Framebuffer 的界面系统开发

嵌入式系统上的 GUI 在实时系统中的地位将越来越重要，这些系统对 GUI 的基本要求包括：轻型、占用资源少；高性能；高可靠性等方面。目前在嵌入式 Linux 系统方面的主要的 GUI 系统包括 Qt/ Embedded、MiniGUI、Nano-X Window System、OpenGU 等。这里介绍中国 Linux 公社论坛上的 EGui 开源项目，让大家初步了解如何开发一个界面。

EGui 是一个支持 Linux Framebuffer 并遵循 LGPL 协议的图形库。EGui 包括三个核心部分，Kernel 驱动部分、Libegui 部分、Libwidget 部分。Kernel 驱动部分主要作用：

- ❑ 获取 Framebuffer 信息。
- ❑ 创建/dev/egui 设备文件号，默认是 c 240 0。
- ❑ 分配窗口的 ID 号。
- ❑ 获取输入事件，并且分发给窗口。
- ❑ 储存窗口的信息，包括位置。

Libegui 部分主要作用：

- ❑ 从/dev/egui 获取和 Kernel 通信设备的 fd 号。
- ❑ 从/dev/fb 获取 Framebuffer 地址。
- ❑ 对显卡的读写实现，包括：点，线，矩形，图片。
- ❑ 窗口的实现，绘制。
- ❑ 光标的移动和绘制。
- ❑ 事件传给 widget。

Libwidget 部分主要作用：

- ❑ 创建不同类型的 widget，例如：button，form，pixmap，edit 等。
- ❑ 采用统一接口实现 widget 的显示，事件，等等。
- ❑ 事件的分发。
- ❑ 父子 widget 机制的实现。

下面重点看看 Libegui 部分。先看打开 Framebuffer 代码：

```
struct fb_fix_screeninfo fb_fix;
struct fb_var_screeninfo fb_var;
```

```
int FBfd;
int eguifd;
Efont defaultfn;
int Egui_open( EGui_FBinfo *fbinfo)
{
  char devfile[12];
  //打开事件设备
  eguifd = open("/dev/egui", O RDONLY);
  if(eguifd < 0)
    {
      printf ("Open /dev/egui error,Don't open display device!\n");
      printf ("Please insmod kegui.ko\n");
      return 1;
    }
  if (-1 == ioctl(eguifd,EGUI GET FBINFO,fbinfo))
    {
      printf("ioctl EGUI_GET_FBINFO error\n\n");
      return 1;
    }
  memcpy (devfile,fbinfo->fbdevfile,8);
  sprintf (devfile,"/dev/fb%d",fbinfo->dev);
 //打开 Framebuffer 设备
  FBfd = open(devfile, O_RDWR);
  if(FBfd < 0)
    {
      printf("Open %s error,Don't open display device!\n", fbinfo->fbdevfile);
      return 1;
    }
  //将物理地址映射为虚拟地址
  fbinfo->Egui_address = (unsigned char *)mmap(NULL, (int)fbinfo->smem_len+(int)
  fbinfo->Egui_phyaddress,
                               PROT_READ | PROT_WRITE, MAP_SHARED,FBfd, 0);
  if (-1L == (long) fbinfo->Egui_address)
    {
      printf("Mmap error! mem:%d \n", fbinfo->Egui_phyaddress);
      return 1;
    }
  //初始化字体
  defaultfn.width     = 8;
  defaultfn.height    = 16;
  defaultfn.font_buf  = fontdata_8x16;
  if (fbinfo->p_bpp == 1)
    set_palette(0, 256, (void *)palette);
  Egui timer init();
  Egui init signal ();
  return 0;
}
```

再看如何进行点操作：

```
Int  Egui_wpixel(int x,int y , Ecolor * color, EGui_Window * ewindow)
{
  int bpp;
  int pwidth;
  int pixel;
  unsigned char * address;
  EGui_FBinfo * fbinfo= ewindow->fbinfo;
  address = fbinfo->Egui address;
  bpp     = fbinfo->p_bpp;
  pwidth  = fbinfo->p_width;
  pixel = color->pixel;
  switch (bpp)
    {
    case 1:          // 8 位
      address [x + (pwidth * y)] = pixel;
      break;
    case 2:          // 16 位
      *(volatile unsigned short *)(address + (x*bpp) + (pwidth*y)) = pixel & 0xFFFF;
      break;
    case 3:          // 24 位
      *(volatile unsigned short *)(address + (x*bpp) + (pwidth*y)) = pixel & 0xFFFF;
      address [(x*bpp) + (pwidth*y) +2] = (pixel >>16) & 0xFF;
      break;
    case 4:          // 32 位
      *(volatile unsigned int *)(address + (x*bpp) + (pwidth*y)) = pixel;
      break;
    }
}
```

画水平直线操作：

```
int Egui_hline (int x,int y,int x1,int y1, Ecolor * color, EGui_Window * ewindow)
{
  int i;
  int maxx,minx;
  EGui_FBinfo * fbinfo= ewindow->fbinfo;
  if (y != y1)
    {
      printf ("No hline y=%d,y1=%d\n",y,y1);
    }
  maxx = x1;
  minx = x;
  if (x>x1)
    {
```

```
        maxx = x;
        minx = x1;
      }
    //判断尺寸是否合法
    if (minx > ewindow->width)
      return -1;
    if (y > ewindow->height)
      return -1;
    minx = ewindow->x + minx;
    y    = ewindow->y + y;
    if (maxx > ewindow->width)
      maxx = ewindow->x + ewindow->width;
    else maxx = ewindow->x + maxx;
    //逐点赋值
    for (i = minx;i <= maxx ; i++)
      {
        Egui_wpixel (i,y,color,ewindow);
      }
    return 0;
  }
```

画垂直直线操作：

```
  int  Egui_vline (int x,int y,int x1,int y1, Ecolor * color, EGui_Window * ewindow)
  {
    int i;
    int maxy,miny;
    EGui_FBinfo * fbinfo= ewindow->fbinfo;
    if (x != x1){
      printf ("No vline x=%d,x1=%d\n",x,x1);
    }
    maxy = y1;
    miny = y;
    if (y>y1){
      maxy = y;
      miny = y1;
    }
    if (miny > ewindow->height)
      return -1;
    if (x > ewindow->width)
      return -1;
    miny = ewindow->y + miny;
    x    = ewindow->x + x;
    if (maxy > ewindow->height)
      maxy = ewindow->y + ewindow->height;
    else maxy = ewindow->y + maxy;
```

```
    //逐点赋值
    for (i=miny;i <= maxy;i++){
      Egui_wpixel (x,i,color,ewindow);
    }
    return 0;
}
```


有上面的函数，画区域就简单了：

```
int Egui_rect (int x,int y,int x1,int y1, Ecolor *color, EGui_Window * ewindow)
{
  int s_width;
  int s height;
  EGui_FBinfo * fbinfo= ewindow->fbinfo;
  s_width  = fbinfo->screen_width;
  s height = fbinfo->screen height;
  if (x<0||x1<0||y<0||y1<0){
    printf ("Egui ERROR:x = %d x1 = %d y = %d y1 = %d,someone <0.\n",x,x1,y,y1);
    return -1;
  }
  //判断坐标是否合法
  if (x >s width-1||x1>s width-1||y>s height-1||y1>s height-1){
    printf ("Egui ERROR:x = %d x1 = %d y = %d y1 = %d,someone too big.\n",x,x1,y,y1);
    return -1;
  }
  //画水平线
  Egui hline (x,y,x1,y,color,ewindow);
  Egui_hline (x,y1,x1,y1,color,ewindow);
  //画垂直线
  Egui_vline (x,y,x,y1,color,ewindow);
  Egui vline (x1,y,x1,y1,color,ewindow);
  return 0;
}
```

最后看看如何在屏幕上绘制字符：

```
Int Egui_drawchar(int x,int y,char ch, Efont *font,Ecolor * color,  EGui_Window *ewindow)
{
  int i,j;
  //绘制字符的图形
  for (j=0;j<font->height;j++)
  {
    for (i=0;i<font->width;i++)
     {
      if(font->font_buf[ch * font->height + j] & (1<<i))
```

```
            {
              Egui_wpixel (x + ewindow->x + font->width - i,
                    (y + j) + ewindow->y,color,ewindow);
            }
        }
    }
}
```

字体使用 struct _Efont 描述：

```
struct  Efont {
  int            width;       //字体宽度，8 个像素
  int            height;      //字体高度，16 个像素
  unsigned char * font_buf;  //字体缓冲地址
};
```

以字符‘7’的字体数据为例子：

```
static unsigned char fontdata_8x16[FONTDATAMAX]=
{
...
    // 55 0x37 '7'
    0x00, // 00000000
    0x00, // 00000000
    0xFE, // 11111110
    0xC6, // 11000110
    0x06, // 00000110
    0x06, // 00000110
    0x0C, // 00001100
    0x18, // 00011000
    0x30, // 00110000
    0x30, // 00110000
    0x30, // 00110000
    0x30, // 00110000
    0x00, // 00000000
    0x00, // 00000000
    0x00, // 00000000
    0x00, // 00000000
...
};
```

当然 EGui 还实现了一些控件和窗口、定时器、事件处理机制等，本文不继续讨论，读者可以从网络上下载代码自己研究。

第 7 章 输入子系统驱动

本章和第 8 章将介绍 Linux 中的输入子系统（Input Subsystem）驱动开发。输入子系统不仅能用来支持鼠标、键盘等输入设备，而且也支持蜂鸣器、LED 等设备。本章将利用 Linux 中的输入子系统（Input Subsystem）提供的方法重新开发第 3 章的键盘与 LED 驱动，并介绍 USB 鼠标的输入设备驱动原理，第 8 章将介绍输入设备驱动形式的触摸屏驱动开发。

7.1 Linux 输入设备驱动

输入设备驱动（input drivers）是 Linux 中为支持所有输入设备而设计的一类驱动。这些驱动与硬件直接对话，且产生键盘和鼠标等输入事件，将事件传递给内核或界面。输入子系统包含在内核中，内核向用户空间提供一个与输入设备无关的统一接口。输入子系统的三个核心元素是输入系统核心、驱动、事件处理，它们之间通过事件进行通信。输入子系统定义在<linux/input.h>中。输入设备的原理如图 7.1 所示。

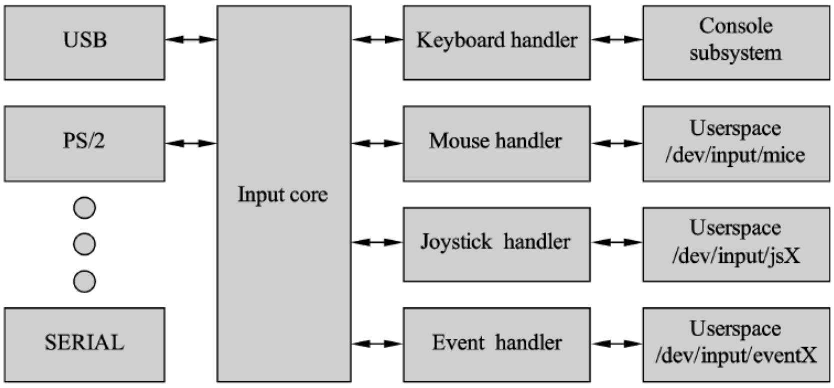


图 7.1 输入设备的原理

事件用下面的结构描述：

```
struct input_event {
    struct timeval time;     //时间戳
    _u16 type;               //驱动类型
    _u16 code;               //事件码
    _s32 value;              //事件值
};
```

输入设备驱动结构：

```
#define NBITS(x) ((((x)-1)/BITS_PER_LONG)+1)
//返回 x 个位的位组的整型长度（多少个 4 字节）
struct input dev
{
    //存放私有数据
    void *private;
    char *name;//名称
    char *phys;
    char *uniq;
    struct input_id id;
    //响应的事件类型
    unsigned long evbit[NBITS(EV_MAX)];
    //响应哪些按键
    unsigned long keybit[NBITS(KEY_MAX)];
    unsigned long relbit[NBITS(REL MAX)];
    unsigned long absbit[NBITS(ABS MAX)];
    unsigned long mscbit[NBITS(MSC MAX)];
    unsigned long ledbit[NBITS(LED MAX)];
    unsigned long sndbit[NBITS(SND MAX)];
    unsigned long ffbit[NBITS(FF_MAX)];
    int ff_effects_max;
    //键盘码表
    unsigned int keycodemax;
    unsigned int keycodesize;
    void *keycode;
    unsigned int repeat key;
    struct timer list timer;
    struct pm_dev *pm_dev;
    struct pt_regs *regs;
    int state;
    int sync;
    int abs[ABS_MAX + 1]; //相对值
    int rep[REP_MAX + 1];//自重复
    unsigned long key[NBITS(KEY_MAX)];
    unsigned long led[NBITS(LED_MAX)];
    unsigned long snd[NBITS(SND_MAX)];
    //相对值的范围
    int absmax[ABS MAX + 1];
    int absmin[ABS MAX + 1];
    int absfuzz[ABS MAX + 1];
    int absflat[ABS MAX + 1];
    //操作函数
    int (*open)(struct input_dev *dev);
```

```
        void (*close)(struct input_dev *dev);
        int (*accept)(struct input_dev *dev, struct file *file);
        int (*flush)(struct input_dev *dev, struct file *file);
        int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
        int (*upload_effect)(struct input_dev *dev, struct ff_effect *effect);
        int (*erase_effect)(struct input_dev *dev, int effect_id);
        struct input_handle *grab;
        struct device *dev;
        struct list_head    h_list;
        struct list_head    node;
    };
```

dev->number 是在驱动注册时由输入系统指定的。dev->id 包含总线 ID（PCI，USB）、厂商 ID 和设备 ID。dev->name 和 dev->id 必须在注册之前设置。keycode 是从扫描码到输入系统输入码的转换表，keycodemax 是码表尺寸，keycodesize 代表每个码所占空间。

下面介绍如何开发一个标准的输入设备。首先确保内核中包含 Event 接口，如图 7.2 所示。

```
<*> Mouse interface
[ ]    Provide legacy /dev/psaux device
(1024)   Horizontal screen resolution
(768)   Vertical screen resolution
< >   Joystick interface
< >   Touchscreen interface
<*>   Event interface
< >   Event debugging
---   Input I/0 drivers
< >   Gameport support
---   Serial i/o support
< >   i8042 PC Keyboard controller
<*>   Serial port line discipline
< >   ct82c710 Aux port controller
---   Input Device Drivers
<*>   Keyboards
```

图 7.2　在内核中添加 Event 接口

输入设备在加载后会在/dev/input 下生成 event 节点。下面是一个简单的输入设备驱动，演示了一个按钮，该按钮通过 BUTTON_PORT 端口输入，当按钮按下后释放，将产生 BUTTON_IRQ 中断：

```
static void button_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    input_report_key(&button_dev, BTN_1, inb(BUTTON_PORT) & 1);
    input_sync(&button_dev);
}
//键盘模块初始化
static int _init button_init(void)
{
    if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
            printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
            return -EBUSY;
       }
    //设置成响应键值
    button_dev.evbit[0] = BIT(EV_KEY);
    //设置响应码表
    button_dev.keybit[LONG(BTN_0)] = BIT(BTN_0);
    //注册标准输入设备
    input_register_device(&button_dev);
}
```

```
//键盘模块结束
static void _exit button_exit(void)
{
    nput_unregister_device(&button_dev);
    free irq(BUTTON IRQ, button interrupt);
}
module_init(button_init);
module_exit(button_exit);
```

初始化函数中先申请一个中断，然后给 button_dev 的 evbit 赋值，这样其他模块就可以知道这个设备所能产生或接收的事件类型了。事件类型包括：

- ❑ EV_KEY：绝对二进制值，如键盘或按钮。
- ❑ EV_REL：相对结果，如鼠标设备。
- ❑ EV_ABS：绝对整数值，如操纵杆或书写板。
- ❑ EV_MSC：其他类。
- ❑ EV_LED：LED 或其他指示设备。
- ❑ EV_SND：声音输出，如蜂鸣器。
- ❑ EV_REP：允许按键自重复。
- ❑ EV_FF：向设备发送强制反馈。
- ❑ EV_FF_STATUS：设备发送强制反馈回复信号。
- ❑ EV_PWR：电源管理事件。

支持自重复只需要在 dev->evbit 中设置 EV_REP，其他的事情输入系统将会完成。button_dev.keybit 代表事件码。

可以使用下面的函数初始化和注册、释放一个输入设备：

```
static inline void init_input_dev(struct input_dev *dev)
void input_register_device(struct input_dev *);
void input_unregister_device(struct input_dev *);
```

在中断中使用下面的函数来报告相应的事件：

```
static inline void input_report_key(struct input_dev *dev, unsigned int code, int value);
static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value);
static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value);
static inline void input_report_ff(struct input_dev *dev, unsigned int code, int value);
```

最基本的事件类型是 EV_KEY。code 可用的码包括 0 到 KEY_MAX。value 是任何非零值都代表按键按下，零代表按键释放。只有当 value 发生变化时，才会将产生的事件上传到内核的输入子系统。

紧接着，需要通知事件接收者发送了一个完整的报告。这在鼠标移动处理中很重要，X，Y 分量不能分开传送，否则会导致错误。

```
static inline void input_sync(struct input_dev *dev);
```

另外在应用层，Linux 操作系统提供了事件接口，用来与内核的输入子系统进行交互。输入

子系统的设备一般在/dev/input 下建立 event 节点。可以通过该节点向输入子系统发送数据或接收来自输入子系统的消息。可以通过 ioctl 命令获取驱动的能力与支持的特性：

```
int version;
ioctl(fd, EVIOCGVERSION, &version);                    //获取版本
struct input_devinfo device_info;
ioctl(fd, EVIOCGID, &device_info);                     //获取设备信息
char name[256]= "Unknown";
ioctl(fd, EVIOCGNAME(sizeof(name)), name)              //获取名称
uint8_t rel_bitmask[REL_MAX/8 + 1];
ioctl(fd, EVIOCGBIT(EV_REL, sizeof(rel_bitmask))       //获取支持的鼠标特性
```

下面的代码演示了如何在应用层获取 LED 驱动支持的事件码：

```
int fd = -1;
 uint8 t led bitmask[LED MAX/8 + 1];
 int yalv;
 //像一般文件一样打开它
 if ((fd = open(argv[1], O_RDONLY)) < 0) {
   perror("evdev open");
   exit(1);
 }
 memset(led bitmask, 0, sizeof(led bitmask));
 //发送获取事件码命令
 if (ioctl(fd, EVIOCGBIT(EV LED, sizeof(led bitmask)), led bitmask) < 0) {
     perror("evdev ioctl");
 }
 printf("Supported LEDs:\n");
 //逐个分析支持的事件码
 for (yalv = 0; yalv < LED_MAX; yalv++) {
     if (test_bit(yalv, led_bitmask)) {
      printf("  LED type 0x%02x ", yalv);
      switch ( yalv)
      {
          case LED NUML :
               printf(" (Num Lock)\n");
               break;
          case LED CAPSL :
               printf(" (Caps Lock)\n");
               break;
          case LED_SCROLLL :
               printf(" (Scroll Lock)\n");
               break;
          case LED_COMPOSE :
               printf(" (Compose)\n");
               break;
```

```
            case LED KANA :
                printf(" (Kana)\n");
                break;
            case LED_SLEEP :
                printf(" (Sleep)\n");
                break;
            case LED_SUSPEND :
                printf(" (Suspend)\n");
                break;
            case LED MUTE :
                printf(" (Mute)\n");
                break;
            case LED_MISC :
                printf(" (Miscellaneous)\n");
                break;
            default:
                printf(" (Unknown LED type: 0x%04hx)\n", yalv);
        }
      }
   }
   close(fd);
```

7.2 键盘输入设备驱动

这里仍然针对 YL2410 开发板上的键盘电路（图 3.7）编写一个驱动。先看两个相关的宏，它们经常用在输入设备驱动中：

```
#define BIT(x)  (1UL<<((x)%BITS_PER_LONG))
//返回第 x 位为 1 其余位为 0 的整数
#define LONG(x)  ((x)/BITS_PER_LONG)
//返回 x 位在位组的整型位置(多少个 4 字节)
```

码表只有 8 个字节：

```
static unsigned char simplekey keycode[0x08] = {
    [0] = KEY 1, [1] = KEY 2,[2] = KEY 3,[3] = KEY 4,
    [4] = KEY 5, [5] = KEY 6,[6] = KEY 7,[7] = KEY 8,
};
```

第一步是注册输入设备：

```
static struct input_dev simplekey_dev;
static unsigned long  polling_jffs=0;
init_input_dev(&simplekey_dev);
```

```
struct timer list polling timer;
//设置为响应键盘
simplekey_dev.evbit[0] = BIT(EV_KEY);// | BIT(EV_REP);
//设置码表
simplekey_dev.keycode = simplekey_keycode;
simplekey_dev.keycodesize = sizeof(unsigned char);
simplekey_dev.keycodemax = ARRAY_SIZE(simplekey_keycode);
//设置名称
simplekey dev.name = simplekey name;
simplekey dev.phys = simplekey phys;
simplekey dev.id.bustype = BUS AMIGA;
for (i = 0; i < 8; i++)
    if (simplekey_keycode[i])
        set_bit(simplekey_keycode[i], simplekey_dev.keybit);
simplekey_dev.id.vendor = 0x0001;
simplekey_dev.id.product = 0x0001;
simplekey_dev.id.version = 0x0100;
//注册输入设备
input register device(&simplekey dev);
//定时器处理
init timer(&polling timer);
polling timer.data = (unsigned long)0;
polling_timer.function = polling_handler;
```

按钮占用了 4 个中断：

```
static int irqArray[4]=
{
    IRQ_EINT0,
    IRQ EINT2,
    IRQ EINT11,
    IRQ EINT19
};
polling_jffs=jiffies;//记录时间
for (i = 0; i <4; i++) {
    if (request_irq(irqArray[i], &simplekey_interrupt, SA_INTERRUPT, "simplekey",
    NULL)) {
            printk("request button irq failed!\n");
            return -1;
    }
}
```

中断处理采用了定时器，在收到中断后延时等待，防止抖动：

```
static irqreturn_t simplekey_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
```

```
    disable irq(IRQ EINT0);
    disable_irq(IRQ_EINT2);
    disable_irq(IRQ_EINT11);
    disable_irq(IRQ_EINT19);
    //启动定时器
    polling_timer.expires = jiffies + HZ/5;
    add_timer(&polling_timer);
    return IRQ HANDLED;
}
```

polling_handler 函数通过读取相应的 I/O 端口寄存器获取键值：

```
int code=-1;
writel(readl(S3C2410_SRCPND)&0xFFFFFFDA,S3C2410_SRCPND);
mdelay(1);
//扫描按键表，根据中断号，找出所按下的按键
writel(readl(S3C2410_GPBDAT)|0x80,S3C2410_GPBDAT);
writel(readl(S3C2410_GPBDAT)&0xFFFFFFBf,S3C2410_GPBDAT);
//检测 GPB[7: 6]=10 时的寄存器
if((readl(S3C2410 GPFDAT)&(1<< 0)) == 0 )
{
    code=0;
    goto IRQ OUT;
}
else if( (readl(S3C2410_GPFDAT)&(1<< 2)) == 0 )
{
    code=2;
    goto IRQ_OUT;
}
else if( (readl(S3C2410 GPGDAT)&(1<< 3)) ==0 )
{
    code=4;
    goto IRQ OUT;
}
else if( (readl(S3C2410 GPGDAT)&(1<<11)) == 0 )
{
    code=6;
    goto IRQ_OUT;
}
writel(readl(S3C2410_SRCPND)&0xFFFFFFDA,S3C2410_SRCPND);
mdelay(1);
writel(readl(S3C2410 GPBDAT)|0x40,S3C2410 GPBDAT);
writel(readl(S3C2410 GPBDAT)&0xFFFFFF7F,S3C2410 GPBDAT);
//检测 GPB[7: 6]=01 时的寄存器
if((readl(S3C2410 GPFDAT)&(1<< 0)) == 0 )
{
```

```
        code=1;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPFDAT)&(1<< 2)) == 0 )
    {
        code=3;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPGDAT)&(1<< 3)) ==0 )
    {
        code=5;
        goto IRQ_OUT;
    }
    else if( (readl(S3C2410_GPGDAT)&(1<<11)) == 0 )
    {
        code=7;
        goto IRQ_OUT;
    }
```

向内核发送键盘事件：

```
IRQ_OUT:
    enable_irq(IRQ_EINT0);
    enable_irq(IRQ_EINT2);
    enable_irq(IRQ_EINT11);
    enable_irq(IRQ_EINT19);
    if(code>=0)
    {
        //避免中断连续出现
        if((jiffies-polling_jffs)>100)
        {
                polling_jffs=jiffies;
                input_report_key(&simplekey_dev, simplekey_keycode[code], 1);
                input_report_key(&simplekey_dev, simplekey_keycode[code], 0);
                input_sync(&simplekey_dev);
#if DEBUG_DRIVER
                printk("key %d\n",code) ;
#endif
    }
    }
//清除数据
writel(readl(S3C2410_GPBDAT)&0xFFFFFF3F,S3C2410_GPBDAT);
```

测试代码如下：

```
void main()
{
```

```
    int fd = -1;
    char name[256]= "Unknown";
    int yalv;
    //打开输入设备
    if ((fd = open("/dev/input/event1", O_RDONLY)) < 0) {
    perror("evdev open");
    exit(1);
    }
    if(ioctl(fd, EVIOCGNAME(sizeof(name)), name) < 0) {
    perror("evdev ioctl");
    }
    //返回的数据字节
    size t rb;
    //每次可读 64 个事件
    struct input event ev[2];
    while(1)
    {
        rb=read(fd,ev,sizeof(struct input_event)*2);
        //数据是否完整
        if (rb < (int) sizeof(struct input_event))
        {
            perror("evtest: short read");
            exit (1);
        }
        printf("read %d event\n",(int)(rb / sizeof(struct input_event)));
        for (yalv = 0;yalv < (int) (rb / sizeof(struct input_event));yalv++)
        {
            //判断是否是需要的事件
            if (EV KEY == ev[yalv].type)
            {
                printf("%ld.%06ld ",ev[yalv].time.tv_sec,ev[yalv].time.tv_usec);
                printf("type %d code %d value %d\n",
                                ev[yalv].type,ev[yalv].code,ev[yalv].value);
            }
        }
    }
    close(fd);
}
```

测试结果如下：

[root@(none) tmp]# insmod button.ko

Using button.ko

initialize button ok![root@(none) tmp]#

[root@(none) tmp]# rz

..[root@(none) tmp]# .**B0100000023be50

[root@(none) tmp]# ls

```
button.ko   demotest   flashdisk   images   mplayer   sdcard   udisk
[root@(none) tmp]# ./demotest
key 2
read 2 event
103.191082 type 1 code 4 value 1
103.191102 type 1 code 4 value 0
read 1 event
key 3
read 2 event
104.562087 type 1 code 5 value 1
104.562109 type 1 code 5 value 0
read 1 event
key 4
read 2 event
111.991103 type 1 code 6 value 1
111.991129 type 1 code 6 value 0
```

7.3 在 MiniGUI 中加入键盘驱动

MiniGUI 最初是魏永明主持的开源界面系统。现在由北京飞漫软件技术有限公司维护和开发，已经可以支持包括 Linux 和 vxworks 在内的多种操作系统。本节基于 MiniGUI1.3.X。MiniGUI 的 IAL 层通过 INPUT 数据结构来表示输入引擎：

```
typedef struct tagINPUT
{
    char*   id;
    //初始化与结束
    BOOL (*init_input) (struct tagINPUT *input, const char* mdev, const char* mtype);
    void  (*term_input)  (void);
    //以下为鼠标操作接口
    int   (*update_mouse)  (void);
    //更新鼠标状态，返回 1 表示成功
    void  (*get_mouse_xy)  (int* x,  int* y);
    //返回鼠标位置，有可能做了适当地边界检查和支持屏幕显示旋转时对坐标的转换
    void  (*set_mouse_xy)  (int x,  int y);
    int   (*get_mouse_button)  (void);
    //返回了鼠标和触摸屏状态，即用户是否触摸了屏幕，或是否按下了左键，或者鼠标哪个键被按下
    void  (*set_mouse_range)  (int minx,  int miny,  int maxx,  int maxy);
    void  (*suspend_mouse)  (void);
    int  (*resume_mouse)  (void);
    //以下为键盘操作接口
    int   (*update_keyboard)  (void);
```

```
    //通知底层键盘更新。根据键盘消息，填充键盘按键状态 state 数组。返回值是 NR_KEYS
    const char* (*get_keyboard_state) (void);
    //获取键盘状态，返回一个包含以扫描码索引的键盘按键状态数组的地址
    void (*suspend_keyboard) (void);
    int (*resume_keyboard) (void);
    void (*set_leds) (unsigned int leds);
#ifdef _LITE_VERSION
    int (*wait_event) (int which, int maxfd, fd_set *in, fd_set *out, fd_set *except,
            struct timeval *timeout);
#else
    int (*wait_event) (int which, fd_set *in, fd_set *out, fd_set *except,
            struct timeval *timeout);
#endif
    wait_event这个函数首先将先前打开的两个设备的文件描述符与传入的in文件描述符集合并在了一
    //起，然后调用了 select 系统调用。当 select 系统调用返回大于 0 的值时，该函数检查在两个文
    //件描述符上是否有可读的数据等待读取，如果是，则分别从两个文件描述符读取触摸屏和按键数据。
    //返回 int 型 retvalue 变量，其中包含了信息：鼠标事件发生、键盘事件发生或者鼠标和键盘事件
    //都发生了（retvalue |= IAL_MOUSEEVENT，retvalue |= IAL_KEYEVENT）
    char mdev [MAX PATH + 1];
}INPUT;
```

应该在引擎的初始化函数中设置 INPUT 数据结构：

```
input->update_mouse = mouse_update;
input->get_mouse_xy = mouse_getxy;
input->set_mouse_xy = NULL;
input->get_mouse_button = mouse_getbutton;
input->set_mouse_range = NULL;
input->wait_event = wait_event;
```

下面介绍如何为 MiniGUI 编写一个键盘的 IAL 文件。键盘就是上面的 2*4 键盘。首先编写一个头文件：

```
#define KEYVALUE_0      0x01
#define KEYVALUE_1      0x02
#define KEYVALUE 2      0x03
#define KEYVALUE 3      0x04
#define KEYVALUE 4      0x05
#define KEYVALUE 5      0x06
#define KEYVALUE 6      0x07
#define KEYVALUE 7      0x08
#define MAX_KEYVALUE    0x08
BOOL   Init2410Input (INPUT* input, const char* mdev, const char* mtype);
Void Term2410Input (void);
```

定义键盘文件描述、键盘映射表：

```
static key fd=-1;
static short KEYCODE=0,KEYSTATUS=0;
static unsigned char keycode_scancode[MAX_KEYVALUE + 1];
```

在初始化的时候打开键盘：

```
BOOL Init2410Input (INPUT* input, const char* mdev, const char* mtype)
{
    char name[256]= "Unknown";
    if ((key fd = open("/dev/input/event1", O RDONLY|O NONBLOCK)) < 0) {
         fprintf (stderr, "2410: Can not open buttonts!\n");
         return FALSE;
    }
    if(ioctl(key_fd, EVIOCGNAME(sizeof(name)), name) < 0) {
         fprintf (stderr, "2410: Can not EVIOCGNAME!\n");
    }
    ...
    init_code_map_nonumlock();
}
```

其中 init_code_map_nonumlock 用来初始化映射表。MiniGUI 中使用 SCANCODE_*来表示键盘扫描码（/include/common.h）。键盘扫描码是键盘值与 ASCII 符号的映射码表，一般都有国际标准。这里将 8 个按键映射成数字键 1～8。

```
static void init_code_map_nonumlock(void)
{
    keycode_scancode[KEYVALUE_0] = SCANCODE_1;
    keycode_scancode[KEYVALUE_1] = SCANCODE_2;
    keycode_scancode[KEYVALUE_2] = SCANCODE_3;
    keycode_scancode[KEYVALUE_3] = SCANCODE_4;
    keycode scancode[KEYVALUE 4] = SCANCODE 5;
    keycode scancode[KEYVALUE 5] = SCANCODE 6;
    keycode scancode[KEYVALUE 6] = SCANCODE 7;
    keycode scancode[KEYVALUE 7] = SCANCODE 8
}
```

wait_event 是 MiniGUI 用来获取键盘、鼠标消息的接口：

```
static int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,struct
timeval *timeout)
{
    fd_set rfds;
    int    retvalue = 0;
    //扫描键盘
    if(which & IAL KEYEVENT)
    {
        struct input_event ev[2];
```

```
            int yalv=0;
            int rb;
            rb=read(key_fd,ev,sizeof(struct input_event));
            if(rb>0)
            {
                if (rb >= (int) sizeof(struct input_event))
                {
                    if (EV_KEY == ev[yalv].type)
                    {
                        //根据 ev[yalv].code 确定，也就是驱动中的返回值
                        KEYCODE=ev[yalv].code-2;
                        KEYSTATUS=ev[yalv].value;
                        retvalue |= IAL_KEYEVENT;
                    }
                }
            }
        }
        //扫描鼠标
        if ((which & IAL_MOUSEEVENT) && ts >= 0) {
        ...
        }
      return retvalue;
    }
```

按下后会调用，修改相应的键盘的扫描表的状态。注意返回值 NR_KEYS 为键盘扫描码的最大系数。

```
    static int keyboard_update(void)
    {
        //更新码表相应键值的状态
        state[keycode_scancode[KEYCODE]] = KEYSTATUS;
        printf("%d,%d\n",keycode_scancode[KEYCODE],KEYSTATUS);
        return NR_KEYS;
    }
```

MiniGUI 通过 keyboard_getstate 获取当前键盘状态，这个函数只要返回状态数组的指针就可以了。

```
    static const char* keyboard_getstate(void)
    {
        return (char *)state;
    }
```

万事俱备后，只要注册引擎就可以了。在 ial.c 中增加：

```
    #ifdef _SMDK2410_IAL
       #include "2410.h"
```

```
#endif
static INPUT inputs [] =
{
#ifdef _SMDK2410_IAL
    {"SMDK2410", Init2410Input, Term2410Input},
#endif
}
```

7.4 LED 输入设备驱动

开发 LED 输入设备其实就是实现一个输入事件处理函数。这里仍然以图 3.6 作为例子。

```
static char s3c2410_LED_name[] = "S3C2410led";
static char s3c2410_LED_phys[] = "s3c2410led";
static struct input_dev s3c2410_LED_dev;
```

注册 LED 输入设备：

```
static int __init s3c2410_LED_init(void)
{
    LED_SI_OUT;//见第 3 章
    s3c2410_LED_dev.evbit[0] = BIT(EV_LED);//类型设置
    s3c2410_LED_dev.ledbit[0] = BIT(LED_NUML);//事件码设置
    s3c2410_LED_dev.event = s3c2410_LED_event;//处理函数设置
    s3c2410_LED_dev.name = s3c2410_LED_name;
    s3c2410_LED_dev.phys = s3c2410_LED_phys;
    s3c2410_LED_dev.id.bustype = BUS_HOST;
    s3c2410_LED_dev.id.vendor = 0x001f;
    s3c2410_LED_dev.id.product = 0x0001;
    s3c2410_LED_dev.id.version = 0x0100;
    input_register_device(&s3c2410_LED_dev);
    printk(KERN_INFO "input: %s\n", s3c2410_LED_name);
    return 0;
}
```

接下来只要实现事件处理函数：

```
static int s3c2410_LED_event(struct input_dev *dev, unsigned int type, unsigned
int code, int value)
{
    printk("s3c2410_LED_event\n");
    if (type != EV_LED)return -1;//类型不对，返回
    switch (code) {
        case LED_NUML:
            break;
```

```
        default:        //事件码不对，返回
            return -1;
    }
    switch(value)//控制 LED
    {
        case 0:
            LED_SI_H;    //见第 3 章
            break;
        case 1:
            LED_SI_L;    //见第 3 章
            break;
    }
    return 0;
}
```

应用层通过/dev/input/event1 操作 LED 灯。注意这里的打开标志设置成 O_WRONLY。

```
int fd = -1;
    char name[256]= "Unknown";
    int yalv;
    //打开事件接口
    if ((fd = open("/dev/input/event1", O WRONLY)) < 0) {
    perror("evdev open");
    exit(1);
    }
    //获取驱动名称
    if(ioctl(fd, EVIOCGNAME(sizeof(name)), name) < 0) {
    perror("evdev ioctl");
    }
    size t rb;
    //填充事件结构
    struct input event ev[2];
    ev[0].type=EV LED;
    ev[0].code=LED NUML;
    ev[0].value=0;
    ev[1].type=EV_LED;
    ev[1].code=LED_NUML;
    ev[1].value=1;
    while(1)
    {
        rb=write(fd,ev,sizeof(struct input_event));
        if (rb < (int) sizeof(struct input event))
        {
            perror("evtest: short write");
            exit (1);
        }
```

```
            sleep(1);
            rb=write(fd,&ev[1],sizeof(struct input_event));
            if (rb < (int) sizeof(struct input_event))
            {
                perror("evtest: short write");
                exit (1);
            }
            sleep(1);
        }
        close(fd);
```

7.5 USB 鼠标输入设备驱动

由图 7.3 可见 USB 鼠标是 USB 设备与输入设备的结合体。以内核中的 USB 鼠标驱动为例子，详细分析 USB 驱动的开发方法。

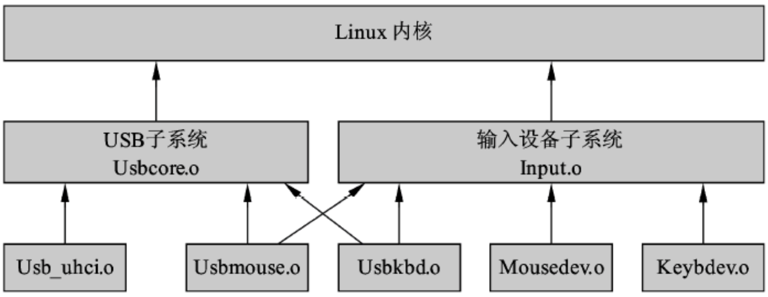


图 7.3　USB 模块之间的相互依赖性

首先定义一个鼠标设备数组：

```
struct usb_mouse {
    char name[128];
    char phys[64];
    struct usb_device *usbdev;//USB 驱动接口
    struct input_dev dev;//输入设备接口
    struct urb *irq;
    int open;
    signed char *data;
    dma_addr_t data_dma;
};
```

在探测函数中处理输入设备的注册：

```
static int usb_mouse_probe(struct usb_interface * intf, const struct usb_device_id * id)
{
    struct usb_device * dev = interface_to_usbdev(intf);
```

```
    struct usb_host_interface *interface;
    struct usb_endpoint_descriptor *endpoint;
    struct usb_mouse *mouse;
    int pipe, maxp;
    char path[64];
    char *buf;
    //指向当前的设置
    interface = intf->cur_altsetting;
    if (interface->desc.bNumEndpoints != 1)
        return -ENODEV;
    //指向接口的第一个端点
    endpoint = &interface->endpoint[0].desc;
    if (!(endpoint->bEndpointAddress & 0x80))
        return -ENODEV;
    if ((endpoint->bmAttributes & 3) != 3)
        return -ENODEV;
    pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
    maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
    if (!(mouse = kmalloc(sizeof(struct usb_mouse), GFP_KERNEL)))
        return -ENOMEM;
    memset(mouse, 0, sizeof(struct usb_mouse));
    mouse->data = usb_buffer_alloc(dev, 8, SLAB_ATOMIC, &mouse->data_dma);
    if (!mouse->data) {
        kfree(mouse);
        return -ENOMEM;
    }
    //分配urb
    mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
    if (!mouse->irq) {
        usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
        kfree(mouse);
        return -ENODEV;
    }
    //设置USB驱动
    mouse->usbdev = dev;
    //输入设备接口设置
    mouse->dev.evbit[0] = BIT(EV_KEY) | BIT(EV_REL);
    mouse->dev.keybit[LONG(BTN_MOUSE)] = BIT(BTN_LEFT) | BIT(BTN_RIGHT) |
    BIT(BTN_MIDDLE);
    mouse->dev.relbit[0] = BIT(REL_X) | BIT(REL_Y);
    mouse->dev.keybit[LONG(BTN_MOUSE)] |= BIT(BTN_SIDE) | BIT(BTN_EXTRA);
    mouse->dev.relbit[0] |= BIT(REL_WHEEL);
    mouse->dev.private = mouse;
    mouse->dev.open = usb_mouse_open;
    mouse->dev.close = usb_mouse_close;
```

```
    usb_make_path(dev, path, 64);
    sprintf(mouse->phys, "%s/input0", path);
    mouse->dev.name = mouse->name;
    mouse->dev.phys = mouse->phys;
    mouse->dev.id.bustype = BUS_USB;
    //厂商设置
    mouse->dev.id.vendor = dev->descriptor.idVendor;
    mouse->dev.id.product = dev->descriptor.idProduct;
    mouse->dev.id.version = dev->descriptor.bcdDevice;
    mouse->dev.dev = &intf->dev;
    if (!(buf = kmalloc(63, GFP_KERNEL))) {
        usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
        kfree(mouse);
        return -ENOMEM;
    }
    if (dev->descriptor.iManufacturer &&
        usb_string(dev, dev->descriptor.iManufacturer, buf, 63) > 0)
            strcat(mouse->name, buf);
    if (dev->descriptor.iProduct &&
        usb_string(dev, dev->descriptor.iProduct, buf, 63) > 0)
            sprintf(mouse->name, "%s %s", mouse->name, buf);
    if (!strlen(mouse->name))
        sprintf(mouse->name, "USB HIDBP Mouse %04x:%04x",
            mouse->dev.id.vendor, mouse->dev.id.product);
    kfree(buf);
    //填充中断型 urb
    usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
             (maxp > 8 ? 8 : maxp),
             usb_mouse_irq, mouse, endpoint->bInterval);
    mouse->irq->transfer_dma = mouse->data_dma;
    mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    //注册鼠标输入设备
    input_register_device(&mouse->dev);
    printk(KERN_INFO "input: %s on %s\n", mouse->name, path);
    usb_set_intfdata(intf, mouse);
    return 0;
}
```

打开函数中提交 urb：

```
static int usb_mouse_open(struct input_dev *dev)
{
    struct usb_mouse *mouse = dev->private;
    if (mouse->open++)
        return 0;
    mouse->irq->dev = mouse->usbdev;
```

```
    if (usb_submit_urb(mouse->irq, GFP_KERNEL)) {
        mouse->open--;
        return -EIO;
    }
    return 0;
}
```


提交 urb 后，如果有数据，则调用回调处理。在回调函数中处理完毕后再次提交 urb。注意在打开函数 usb_mouse_open 中用的是 GFP_KERNEL 标志，在 usb_mouse_irq 中断中必须使用 SLAB_ATOMIC（等于 GFP_ATOMIC）。

```
static void usb_mouse_irq(struct urb *urb, struct pt_regs *regs)
{
    struct usb_mouse *mouse = urb->context;
    signed char *data = mouse->data;
    struct input_dev *dev = &mouse->dev;
    int status;
    switch (urb->status) {
    case 0:
        break;
    case -ECONNRESET:
    case -ENOENT:
    case -ESHUTDOWN:
        return;
    default:
        goto resubmit;
    }
    input_regs(dev, regs);
    //向输入子系统发送鼠标事件
    input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
    input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
    input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
    input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
    input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);
    input_report_rel(dev, REL_X,     data[1]);
    input_report_rel(dev, REL_Y,     data[2]);
    input_report_rel(dev, REL_WHEEL, data[3]);
    input_sync(dev);
resubmit:
    status = usb_submit_urb (urb, SLAB_ATOMIC);
    if (status)
        err ("can't resubmit intr, %s-%s/input0, status %d",
                mouse->usbdev->bus->bus_name,
                mouse->usbdev->devpath, status);
}
```

测试程序如下：

```
void main()
{
    int fd = -1;
    char name[256]= "Unknown";
    int yalv;
    if ((fd = open("/dev/input/event1", O RDONLY)) < 0) {
    perror("evdev open");
    exit(1);
    }
    if(ioctl(fd, EVIOCGNAME(sizeof(name)), name) < 0) {
    perror("evdev ioctl");
    }
    //返回读取的字节数量
    size_t rb;
    struct input event ev[2];
    while(1)
    {
        //读鼠标事件
        rb=read(fd,ev,sizeof(struct input_event)*2);
        if (rb < (int) sizeof(struct input_event))
        {
            perror("evtest: short read");
            exit (1);
        }
        for (yalv = 0;yalv < (int) (rb / sizeof(struct input_event));yalv++)
        {
            printf("read %d event\n",ev[yalv].type);
            if (EV REL == ev[yalv].type)
            {
                printf("%ld.%06ld ",ev[yalv].time.tv sec,ev[yalv].time.tv usec);
                printf("type %d code %d value %d\n",
                                ev[yalv].type,ev[yalv].code,ev[yalv].value);
            }
            if (EV_KEY == ev[yalv].type)
            {
                printf("%ld.%06ld ",ev[yalv].time.tv_sec,ev[yalv].time.tv_usec);
                printf("type %d code %d value %d\n",
                                ev[yalv].type,ev[yalv].code,ev[yalv].value);
            }
        }
    }
    close(fd);
}
```

下面是测试结果：

```
read 1 event
108.702099 type 1 code 273 value 1        //右键按下
read 0 event
read 2 event
108.789995 type 2 code 1 value 9          //鼠标移动
read 0 event
read 1 event
108.862059 type 1 code 273 value 0        //右键释放
read 0 event
read 1 event
111.340048 type 1 code 272 value 1        //左键按下
read 0 event
read 1 event
111.518003 type 1 code 272 value 0        //左键释放
read 0 event
read 2 event
113.740016 type 2 code 8 value 1          //中间键上滚
read 0 event
read 2 event
115.013974 type 2 code 8 value -1         //中间键下滚
read 0 event
read 2 event
```

第8章 触摸屏驱动

Linux 中的触摸屏驱动，可以作为一种普通的字符型设备，也可以纳入输入子系统的框架。触摸屏与鼠标的最大区别在于前者是基于绝对的坐标，后者是基于相对的坐标。这意味着在应用层使用触摸屏驱动必须进行校准。通常界面系统会包含一个触摸屏中间驱动层，这是界于触摸屏驱动和界面系统之间的驱动，校准一般是在这个中间层实现的。本章的主要内容包括触摸屏驱动开发、校准与 MiniGUI 中间驱动层编写。

8.1 触摸屏原理

触摸屏附着在显示器的表面，与显示器相配合使用，如果能测量出触摸点在屏幕上的坐标位置，则可根据显示屏上对应坐标点的显示内容获知触摸者的意图。按照触摸屏的工作原理和传输信息的介质，可以把触摸屏分为 4 种，它们分别为电阻式、电容感应式、红外线式以及表面声波式。每一类触摸屏都有其各自的优缺点，其中电阻式触摸屏在嵌入式系统中用得较多。下面简要介绍电阻式触摸屏的工作原理。

电阻式触摸屏利用压力感应进行控制。电阻式触摸屏的主要部分是一块与显示器表面非常配合的电阻薄膜屏，这是一种多层的复合薄膜，它以一层玻璃或硬塑料平板作为基层，表面涂有一层透明氧化金属（透明的导电电阻）导电层，上面再盖有一层外表面硬化处理的光滑防擦的塑料层。中间是两层金属导电层，分别在基层之上和塑料层内表面。它的内表面也涂有一层涂层，在它们之间有许多细小的（小于 1/1000 英寸）的透明隔离点把两层导电层隔开绝缘。当手指触摸屏幕时，两层导电层在触摸点位置就有了接触，电阻发生变化，在 X 和 Y 两个方向上产生信号，然后送触摸屏控制器。控制器侦测到这一接触并计算出（X，Y）的位置，再根据这个位置模拟鼠标的运作，触摸屏的触摸示意图如图 8.1 所示。

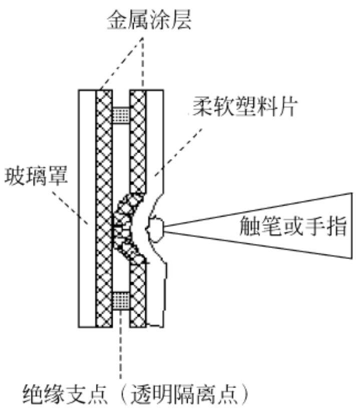


图 8.1　触摸屏的触摸示意图

触摸屏的两个金属导电层是触摸屏的两个工作面，在每个工作面的两端各涂有一条银胶，称为该工作面的一对电极，若在一个工作面的电极对上施加电压，则在该工作面上就会形成均匀连续的平行电压分布。当在 X 方向的电极对上施加一确定的电压，而 Y 方向电极对上不加电压时，在 X 平行电压场中，触点处的电压值可以在 Y+（或 Y–）电极上反映出来，通过测量 Y+电极对地的电压大小，便可得知触点的 X 坐标值。同理，当在 Y 电极对上加电压，而 X 电极对上不加电压时，

通过测量 X+电极的电压，便可得知触点的 Y 坐标。电阻式触摸屏有四线和五线两种。四线式触摸屏的 X 工作面和 Y 工作面分别加在两个导电层上，共有 4 根引出线，分别连到触摸屏的 X 电极对和 Y 电极对上。五线式触摸屏把 X 工作面和 Y 工作面都加在玻璃基层的导电涂层上，但工作时，仍是分时加电压的，即让两个方向的电压场分时工作在同一工作面上，而外导电层则仅仅用来充当导体和电压测量电极。

8.2 S3C2410X 触摸屏控制器

S3C2410X 支持触摸屏接口，它是与 ADC 转换器共用的，由一个触摸屏面板、4 个外部电阻、一个外部电压源、AIN[7]和 AIN[5]组成，S3C2410X 触摸屏接口如图 8.2 所示。

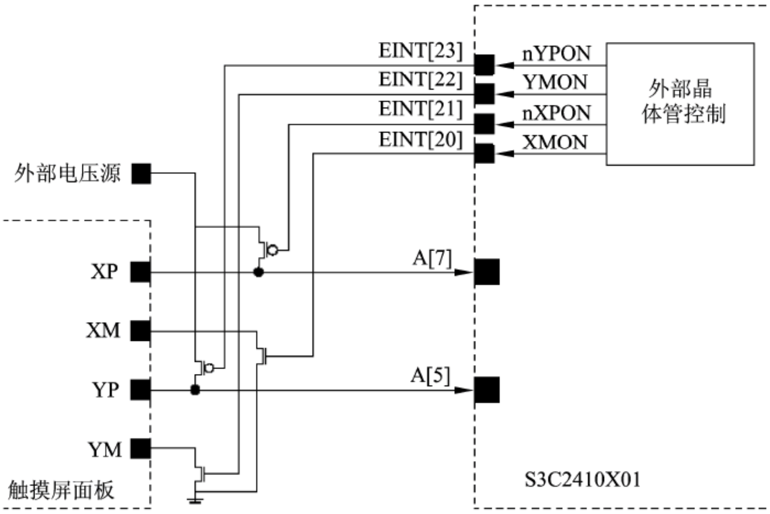


图 8.2 S3C2410X 触摸屏接口

在上面的例子中，AIN[7]接面板 XP 脚，AIN[5]接面板 YP 脚。控制信号 nYPON，YMON，nXPON 以及 XMON 与 4 个外部电阻相连。触摸屏接口有 4 个工作模式，依 ADCTSC 寄存器的 AUTO_PST 和 XY_PST 确定。4 个工作模式的比较表如表 8.1 所示。

表 8.1 触摸屏接口的 4 个工作模式

模式	AUTO_PST	XY_PST	说明
常规转换模式	AUTO_PST=0	XY_PST=0	通用的 ADC 转换
XY 坐标独立转换模式	AUTO_PST=0	XY_PST = 1	转换 X 坐标到 ADCDAT0 的 XPDATA，并产生 INT_ADC 中断
		XY_PST = 2	转换 Y 坐标到 ADCDAT1 的 YPDATA 并产生 INT_ADC 中断
XY 坐标自动转换模式	AUTO_PST=1	XY_PST=0	自动转换 X 坐标到 ADCDAT0 的 XPDATA；同时转换 Y 坐标到 ADCDAT1 的 YPDATA；并产生 INT_ADC 中断

续表

模式	AUTO_PST	XY_PST	说明
等待中断模式		XY_PST=3	当触摸笔点击时产生中断，此时可以采用 XY 坐标独立转换模式或 XY 坐标自动转换模式读取坐标数据

与触摸屏相关的寄存器：

1. ADCON 寄存器

关于 ADCON 寄存器的相关数据如表 8.2 所示。

表 8.2　ADCON 寄存器描述

ADCON	位	描述	初始值
ECFLG	[15]	AD 转换结束标志： 0=AD 转换进行中； 1=AD 转换结束	0
PRSCEN	[14	AD 转换预分频器使能： 0=停止； 1=使能	0
PRSCVL	[13:6]	AD 转换预分频器数值 范围：1～255； 除数为（PRSCVL+1）； 注意 ADC 频率应该设置成小于 PCLK 的 5 倍	0xFF
SEL_MUX	[5:3]	模拟输入通道选择： [0, 7]对应 AIN0～AIN7	0
STDBM	[2]	Standby 模式选择： 0=普通模式； 1=standby 模式	1
READ_START	[1]	通过读取来启动 AD 转换： 0=停止通过读取来启动 AD 转换； 1=使能通过读取来启动 AD 转换	0
ENABLE_START	[0]	启动 AD 操作： 如果 READ_START=1，则这个值无效。 0=无操作； 1=启动 AD 转换，启动后该位清零	0

2. ADCTSC 寄存器

关于 ADCTSC 寄存器的相关数据如表 8.3 所示。

表 8.3　ADCTSC 寄存器描述

ADCTSC	位	描述	初始值
保留	[8]	应该为 0	0
YM_SEN	[7]	选择 YMON 的输出值： 0=YMON 输出 0（YM=Hi-Z）； 1=YMON 输出 1（YM=GND）	0

续表

ADCTSC	位	描述	初始值
YP_SEN	[6]	选择 nYPON 的输出值： 0=nYPON 输出 0（YP=External voltage）； 1=nYPON 输出 1（YP 连接 AIN[5]）	1
XM_SEN	[5]	选择 XMON 的输出值： 0=XMON 输出 0（XM=Hi_Z）； 1=XMON 输出 1（XM=GND）	0
XP_SEN	[4]	选择 nXPON 的输出值： 0=nXPON 输出 0（XP=External voltage）； 1=nXPON 输出 1（XP 连接 AIN[7]）	1
PULL_UP	[3]	上拉开关： 0=允许 XP 上拉； 1=禁止 XP 上拉	1
AUTO_PST	[2]	模式选择： 0=正常 ADC 转换； 1=自动 XY 坐标转换	0
XY_PST	[1:0]	测量模式： 00=无操作； 01=X 坐标测量； 10=Y 坐标测量； 11=等待中断模式	0

3. ADCDLY 寄存器

关于 ADCDLY 寄存器的相关数据如表 8.4 所示。

表 8.4　ADCDLY 寄存器描述

ADCDLY	位	描述	初始值
DELAY	[15:0]	常规转换模式、XY 坐标自动转换模式、XY 坐标自动转换模式下： X/Y 坐标转换延时； 等待中断模式下：触摸笔按下以后产生中断的时间间隔； 注意不能使用 0 值	0x00ff

4. ADCDAT0 寄存器

关于 ADCDAT0 寄存器的相关数据如表 8.5 所示。

表 8.5　ADCDAT0 寄存器描述

ADCDAT0	位	描述	初始值
UPDOWN	[15]	等待中断模式下触摸笔状态： 0=触摸笔按下； 1=触摸笔拿起	-
AUTO_PST	[14]	0=常规 ADC 转换； 1=XY 坐标顺序测量	-

续表

ADCDAT0	位	描述	初始值
XY_PST	[13:12]	00=无操作； 01=X 坐标测量； 10=Y 坐标测量； 11=等待中断模式	-
Reserved	[11:10]	Reserved	
XPDATA	[9:0]	X 坐标值或常规 ADC 转换值，支持到 0x3FF	-

5. ADCDAT1 寄存器

关于 ADCDAT1 寄存器的相关数据如表 8.6 所示。

表 8.6　ADCDAT1 寄存器描述

ADCDAT1	位	描述	初始值
UPDOWN	[15]	等待中断模式下触摸笔状态： 0=触摸笔按下； 1=触摸笔拿起	-
AUTO_PST	[14]	0=常规 ADC 转换； 1=XY 坐标顺序测量	-
XY_PST	[13:12]	00=无操作； 01=X 坐标测量； 10=Y 坐标测量； 11=等待中断模式	-
Reserved	[11:10]	Reserved	
YPDATA	[9:0]	Y 坐标值，支持到 0x3FF	-

8.3　S3C2410X 触摸屏驱动设计

触摸屏可以归为 Linux 标准输入设备。下面依然利用 Linux 输入设备接口来设计触摸屏驱动。电路原理图见图 8.2。首先定义设备结构：

```
struct demo_ts
{
     struct input_dev  tsdev;
     long xp;
     long yp;
     int count;
};
static struct demo ts tsdemo dev;
static char *tsdemo_name = "tsdemo";
static char *tsdemo_phys = "input0";
```

初始化触摸屏接口：

```
void init ts(void)
{
    _raw_writel(_raw_readl(S3C2410_GPGCON)|(0xFF<<24), S3C2410_GPGCON);
    _raw_writel(30000, S3C2410_ADCDLY);
    _raw_writel(0,S3C2410_ADCCON);
    _raw_writel(WAIT4INT(0), S3C2410_ADCTSC);
}
```

初始化输入设备：

```
static int  init tsdemo init(void)
{
     init_ts();
     //申请 ADC 转换中断
    if (request irq(IRQ ADC, adcdemo interrupt, SA INTERRUPT, "tsdemo",NULL)) {
        printk(KERN ERR "s3c2440 ts.c: Could not allocate ts IRQ ADC !\n");
        return -EIO;
    }
    //申请触摸屏中断
    if (request_irq(IRQ_TC, tsdemo_interrupt, SA_INTERRUPT, "tsdemo", NULL)) {
        disable irq(IRQ ADC);
        free irq(IRQ ADC,adcdemo interrupt);
        return -EIO;
    }
    //初始化输入设备
    init input dev(&tsdemo dev.tsdev);
   //接受键值与绝对量
    tsdemo dev.tsdev.evbit[0] = BIT(EV SYN) | BIT(EV KEY) | BIT(EV ABS);
    tsdemo_dev.tsdev.keybit[LONG(BTN_TOUCH)] = BIT(BTN_TOUCH);
    input_set_abs_params(&tsdemo_dev.tsdev, ABS_X, 0, 0x3FF, 0, 0);
    input_set_abs_params(&tsdemo_dev.tsdev, ABS_Y, 0, 0x3FF, 0, 0);
    input set abs params(&tsdemo dev.tsdev, ABS PRESSURE, 0, 1, 0, 0);
    //私有数据
    tsdemo dev.tsdev.private = &tsdemo dev;
    tsdemo_dev.tsdev.name = tsdemo_name;
    tsdemo_dev.tsdev.phys = tsdemo_phys;
    tsdemo_dev.tsdev.id.bustype = BUS_RS232;
    tsdemo dev.tsdev.id.vendor = 0xDEAD;
    tsdemo dev.tsdev.id.product = 0xBEEF;
    tsdemo_dev.tsdev.id.version = 0x0101;
    //注册输入设备
    input_register_device(&tsdemo_dev.tsdev);
    printk(KERN INFO "input: %s\n", tsdemo name);
    return 0;
}
```

退出过程就是释放中断和输入设备：

```
static void _exit tsdemo_exit(void)
{
    free irq(IRQ TC,tsdemo interrupt);
    input unregister device(&tsdemo dev.tsdev);
}
```

下面处理 IRQ_TC 中断：

```
static irqreturn_t tsdemo_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    unsigned long data0;
    unsigned long data1;
    int updown;
    //获取触摸笔状态
    data0 = readl(S3C2410 ADCDAT0);
    data1 = readl(S3C2410 ADCDAT1);
    updown = (!(data0 & S3C2410 ADCDAT0 UPDOWN))
                    && (!(data1 & S3C2410 ADCDAT0 UPDOWN));
    printk("tsdemo interrupt data0 = %ld, data1 = %ld updown = %d\n", data0, data1,
    updown);
   //点击
    if (updown)
    {
        init_timer(&simple_timer);
        simple_timer.function = &touch_timer_fire;
        simple timer.expires = jiffies + SIMPLE TIMER DELAY;
        add timer (&simple timer);
    }
    return IRQ HANDLED;
}
```

通过定时器激发 X 轴的转换，并产生 IRQ_ADC 中断：

```
static void touch_timer_fire(unsigned long data)
{
    del_timer(&simple_timer);
    tsdemo dev.xp = 0;
    tsdemo dev.yp = 0;
    tsdemo dev.count = 0;
     raw writel(0x69, S3C2410 ADCTSC);
     raw writel(0x4c7a, S3C2410 ADCCON);
    usleep(1);}
```

当 tsdemo_dev.count = 0，先获取 X 轴的坐标，并激发 Y 轴的转换，并产生 IRQ_ADC 中断。当 tsdemo_dev.count = 1，则获取 Y 轴的坐标。

```
static irqreturn_t adcdemo_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    if(tsdemo_dev.count==0)
    {
        tsdemo_dev.count= 1;
        _raw_writel(_raw_readl(S3C2410_ADCCON)&(~2), S3C2410_ADCCON);
        tsdemo_dev.yp = _raw_readl(S3C2410_ADCDAT1)
                              & S3C2410 ADCDAT0 XPDATA MASK;
         raw writel(0x9a, S3C2410 ADCTSC);
          raw writel(0x4c6a, S3C2410 ADCCON);
         usleep(1);
    }
    else if(tsdemo_dev.count==1)
    {
       tsdemo_dev.count= 2;
       _raw_writel(_raw_readl(S3C2410_ADCCON)&(~2), S3C2410_ADCCON);
       tsdemo_dev.xp = _raw_readl(S3C2410_ADCDAT0)
                                   & S3C2410_ADCDAT0_XPDATA_MASK;
        printk("PEN DOWN: x: %ld, y: %ld\n",tsdemo dev.xp,tsdemo dev.yp);
       //汇报绝对坐标
        input report abs(&tsdemo dev.tsdev, ABS X, tsdemo dev.xp);
        input report abs(&tsdemo dev.tsdev, ABS Y, tsdemo dev.yp);
       //汇报压力
        //input report key(&tsdemo dev.tsdev, BTN TOUCH, 1);
        input_report_abs(&tsdemo_dev.tsdev, ABS_PRESSURE, 1);
        input_report_abs(&tsdemo_dev.tsdev, ABS_PRESSURE, 0);
        input_sync(&tsdemo_dev.tsdev);
        mdelay(1);
      //允许中断
        writel(WAIT4INT(0), S3C2410_ADCTSC);
    }
    return IRQ HANDLED;
}
```

测试程序如下：

```
void main()
{
    int fd = -1;
    char name[256]= "Unknown";
    int yalv;
    //打开输入设备
    if ((fd = open("/dev/input/event1", O RDONLY)) < 0) {
    perror("evdev open");
    exit(1);
    }
```

```
    if(ioctl(fd, EVIOCGNAME(sizeof(name)), name) < 0) {
    perror("evdev ioctl");
    }
    //返回的数据字节
    size_t rb;
    struct input event ev[5];
    while(1)
    {
        //读鼠标事件
        rb=read(fd,ev,sizeof(struct input_event)*5);
        if (rb < (int) sizeof(struct input event))
        {
            perror("evtest: short read");
            exit (1);
        }
        printf("read %d event\n",(int)(rb / sizeof(struct input_event)));
        for (yalv = 0;yalv < (int) (rb / sizeof(struct input event));yalv++)
        {
            if (EV_ABS == ev[yalv].type)
            {
                printf("%ld.%06ld ",ev[yalv].time.tv_sec,ev[yalv].time.
                tv_usec);
                printf("type %d code %d value %d\n",
                                ev[yalv].type,ev[yalv].code,ev[yalv].value);
            }
        }
    }
    close(fd);
}
```

测试结果：

#insmod demo.ko

#demotest

tsdemo_interrupt data0 = 12935, data1 = 12688 updown = 1

PEN DOWN: x: 572, y: 400

read 2 event

745.572432 type 3 code 0 value 572

745.572462 type 3 code 1 value 400

上面 code=0 代表 X 轴，code=1 代表 Y 轴。

8.4 校准原理及编程思路

图 8.3 是 LCD 坐标和触摸屏的物理坐标的比较。传统的鼠标是一种相对定位系统，只和前一

次鼠标的位置坐标有关。而触摸屏则是一种绝对坐标系统，要选哪就直接点哪，与相对定位系统有着本质的区别。绝对坐标系统的特点是每一次定位坐标与上一次定位坐标没有关系，每次触摸的数据通过校准转为屏幕上的坐标，不管在什么情况下，触摸屏这套坐标在同一点的输出数据都是稳定的。不过由于技术原理的原因，并不能保证同一点在每一次触摸中采样数据相同，不能保证绝对坐标定位，也就是坐标漂移，这是触摸屏最怕出现的问题。对于性能质量好的触摸屏来说，漂移的情况出现并不是很严重，可以采用简单的线性方程校准。

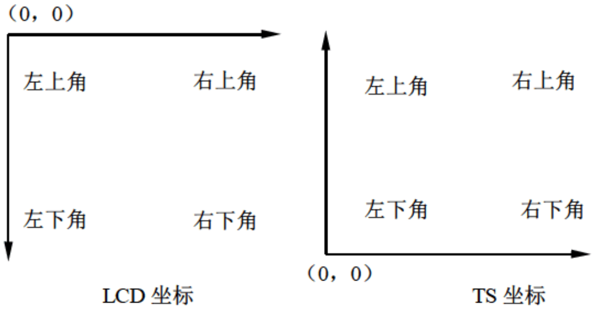


图 8.3　LCD 坐标与触摸屏坐标的区别

8.4.1　线性校准原理

假设 LCD 的宽度为 width，高度为 height。在屏幕左上角采样的坐标值为（$x1$，$y1$），右下角采样的坐标值为（$x2$，$y2$）。可以通过下面的方程求出触摸屏上任意点（x，y）对应的 LCD 上的坐标（X，Y）：

$$X = (x - x1) \times \text{width} / (x2 - x1) \tag{8-1}$$

$$Y = (y - y1) \times \text{height} / (y2 - y1) \tag{8-2}$$

8.4.2　三点校准原理

对于电阻式的触摸屏，许多原因会导致坐标错误。最重要的因素是电子噪声、机械上未对准、缩放因素等。用户有时也是一个重要的因素，例如手指或压屏设备没有保持连续的接触和压力。所有的错误使数据不可靠，必须采用补偿算法加以纠正。

考虑图 8.4 左图的变形，圆形代表 LCD 上的显示的图像，反映到触摸屏上却是一种椭圆。校准算法的难点在于如何将触摸屏上的坐标系转换成显示的图像坐标。

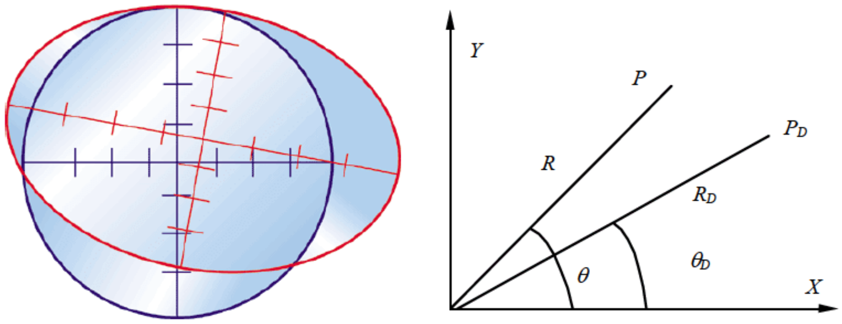


图 8.4　三点校准原理

用矢量 P_D 代表 LCD 上的点，矢量 P 代表 P_D 对应于触摸屏上的点。假设它们之间可用如下公式进行转换：

$$P_D = M_P \tag{8-3}$$

其中 M 是一个合适的转换矩阵。每个点用（x，y）表示，（x，y）依赖于矢量的半径和角度：

$$P_D = [X_D, Y_D] = [R_D \cos\theta_D, R_D \sin\theta_D] \tag{8-4}$$

$$P = [X, Y] = [R \cos\theta, R \sin\theta] \tag{8-5}$$

如果 P 点由于机械未对准导致了角度上的偏移 θ_r：

$$P' = [R \cos(\theta+\theta_r), R \sin(\theta+\theta_r)] \tag{8-6}$$

如果由于某种原因导致 X，Y 方向上有所缩放，缩放系数为 K_x，K_y：

$$P'' = [K_x R \cos(\theta+\theta_r), K_y R \sin(\theta+\theta_r)] \tag{8-7}$$

最后考虑到变换因素，得到最终的点 P_D：

$$P''' = [K_x R \cos(\theta+\theta_r) + X_T, K_y R \sin(\theta+\theta_r) + Y_T] \tag{8-8}$$

角度上的偏移 θ_r 非常小，所以可以假定 $\sin(\theta_r)= \theta_r$，$\cos(\theta_r)=1.0$：

$$\cos(\theta+\theta_r) \approx (\cos\theta - \theta_r \sin\theta) \tag{8-9}$$

$$\sin(\theta+\theta_r) \approx (\sin\theta + \theta_r \cos\theta) \tag{8-10}$$

式（8-8）可以等效于：

$$P_D = [K_x R\cos\theta - \theta_r K_x R\sin\theta + X_T, K_y R\sin\theta + \theta_r K_y R\cos\theta + Y_T] \tag{8-11}$$

根据式（8-5）有：

$$P_D = [K_x X - \theta_r K_x Y + X_T, \theta_r K_y X + K_y Y + Y_T] \tag{8-12}$$

也就是：

$$X_D = AX + BY + C \tag{8-13}$$

$$Y_D = DX + EY + F \tag{8-14}$$

记住，上面的公式在偏移角度很小的情况下成立。可见，只要确定 A、B、C、D、E、F 这 6 个参数就可以得到变换公式，至少需要三个点，才能解出上面的方程。通常选择相距较远的点作为校准点，如图 8.5 所示。

假设上图触摸屏上三点用（X_{D0}，Y_{D0}）、（X_{D1}，Y_{D1}）、（X_{D2}，Y_{D2}）表示，LCD 上的三点用（X_0，Y_0）、（X_1，Y_1）、（X_2，Y_3）。

$$X_{D0} = AX_0 + BY_0 + C \tag{8-15}$$

$$X_{D1} = AX_1 + BY_1 + C \tag{8-16}$$

$$X_{D2} = AX_2 + BY_2 + C \tag{8-17}$$

$$Y_{D0} = DX_0 + EY_0 + F \tag{8-18}$$

$$Y_{D1} = DX_1 + EY_1 + F \tag{8-19}$$

$$Y_{D2} = DX_2 + EY_2 + F \tag{8-20}$$

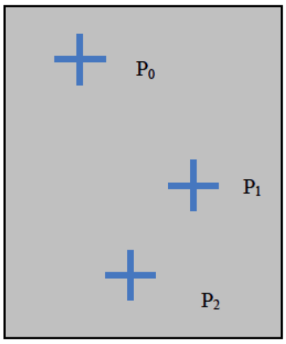


图 8.5　三点校准的三点

由上面的方程得到：

$$K = (X_0 - X_2)(Y_1 - Y_2) - (X_1 - X_2)(Y_0 - Y_2) \tag{8-21}$$

$$A = \frac{((X_{D0} - X_{D2})(Y_1 - Y_2) - (X_{D1} - X_{D2})(Y_0 - Y_2))}{K} \tag{8-22}$$

$$B=\frac{((X_{D1}-X_{D2})(X_0-X_2)-(X_{D0}-X_{D2})(X_1-X_2))}{K} \quad (8\text{-}23)$$

$$C=\frac{(Y_0(X_2X_{D1}-X_1X_{D2})+Y_1(X_0X_{D2}-X_2X_{D0})+Y_2(X_1X_{D0}-X_0X_{D1}))}{K} \quad (8\text{-}24)$$

$$D=\frac{((Y_{D0}-Y_{D2})(Y_1-Y_2)-(Y_{D1}-Y_{D2})(Y_0-Y_2))}{K} \quad (8\text{-}25)$$

$$E=\frac{((Y_{D1}-Y_{D2})(X_0-X_2)-(Y_{D0}-Y_{D2})(X_1-X_2))}{K} \quad (8\text{-}26)$$

$$F=\frac{(Y_0(X_2Y_{D1}-X_1Y_{D2})+Y_1(X_0Y_{D2}-X_2Y_{D0})+Y_2(X_1Y_{D0}-X_0Y_{D1}))}{K} \quad (8\text{-}27)$$

8.5 利用 tslib 库校准

tslib 是一个触摸屏的库，它提供诸如滤波、去抖、校准之类的功能，为不同的触摸屏提供了一个统一的接口。本文介绍的是 tslib1.3 版。

```
struct tsdev {
    int fd;//触摸屏设备文件描述符
    struct tslib_module_info *list;//插件指针
};
```

采样点表示为：

```
struct ts_sample {
    int      x;//X 坐标
    int      y;//Y 坐标
    unsigned int    pressure;//按键轻重
    struct timeval  tv;//发生时间
};
```

tslib/tests/ts_calibrate.c 中提供了校准的例子：

```
int perform_calibration(calibration *cal) {
    int j;
    float n, x, y, x2, y2, xy, z, zx, zy;
    float det, a, b, c, e, f, i;
    float scaling = 65536.0;
    // 矩阵求和
    n = x = y = x2 = y2 = xy = 0;
    for(j=0;j<5;j++) {
        n += 1.0;
        x += (float)cal->x[j];
        y += (float)cal->y[j];
        x2 += (float)(cal->x[j]*cal->x[j]);
```

```
            y2 += (float)(cal->y[j]*cal->y[j]);
            xy += (float)(cal->x[j]*cal->y[j]);
        }
       //得到矩阵行列式，检查行列式是否太小
        det = n*(x2*y2 - xy*xy) + x*(xy*y - x*y2) + y*(x*xy - y*x2);
        if(det < 0.1 && det > -0.1) {
            printf("ts_calibrate: determinant is too small -- %f\n",det);
            return 0;
        }
       //求逆矩阵
        a = (x2*y2 - xy*xy)/det;
        b = (xy*y - x*y2)/det;
        c = (x*xy - y*x2)/det;
        e = (n*y2 - y*y)/det;
        f = (x*y - n*xy)/det;
        i = (n*x2 - x*x)/det;
       // X 坐标求和
        z = zx = zy = 0;
        for(j=0;j<5;j++) {
            z += (float)cal->xfb[j];
            zx += (float)(cal->xfb[j]*cal->x[j]);
            zy += (float)(cal->xfb[j]*cal->y[j]);
        }
       //得到 X 轴校准值
        cal->a[0] = (int)((a*z + b*zx + c*zy)*(scaling));
        cal->a[1] = (int)((b*z + e*zx + f*zy)*(scaling));
        cal->a[2] = (int)((c*z + f*zx + i*zy)*(scaling));
        printf("%f %f %f\n",(a*z + b*zx + c*zy), (b*z + e*zx + f*zy), (c*z + f*zx +
        i*zy));
       // Y 坐标求和
        z = zx = zy = 0;
        for(j=0;j<5;j++) {
            z += (float)cal->yfb[j];
            zx += (float)(cal->yfb[j]*cal->x[j]);
            zy += (float)(cal->yfb[j]*cal->y[j]);
        }
        //得到 Y 轴校准值
        cal->a[3] = (int)((a*z + b*zx + c*zy)*(scaling));
        cal->a[4] = (int)((b*z + e*zx + f*zy)*(scaling));
        cal->a[5] = (int)((c*z + f*zx + i*zy)*(scaling));
        printf("%f %f %f\n",(a*z + b*zx + c*zy), (b*z + e*zx + f*zy), (c*z + f*zx +
        i*zy));
      //完成，返回缩放因子
        cal->a[6] = (int)scaling;
        return 1;
    }
```

该程序先获取屏幕上5个点的坐标，然后进行校准。获取触摸点的坐标是调用 ts_read_raw 函数，ts_read_raw 函数包含两种获取触摸点的坐标的方法，一是通过标准的输入设备接口，另外一种是通过非标准的输入设备接口的方式。下面看看采用标准的输入设备接口的代码：

```
#ifdef USE_INPUT_API
    // 警告：这些静态变量是 tsdev 结构的组成部分吗？
    static int curr_x = 0, curr_y = 0, curr_p = 0;
    static int got_curr_x = 0, got_curr_y = 0;
    int got_curr_p = 0;
    int next_x, next_y;
    int got_next_x = 0, got_next_y = 0;
    int got_tstamp = 0;
    while (total < nr) {
        //开始读标准输入事件
        ret = read(ts->fd, &ev, sizeof(struct input_event));
        if (ret < sizeof(struct input_event)) break;
       printf("ts_calibrate read\n");
       //正常的事件顺序是 ABS_X -> ABS_Y -> ABS_PRESSURE
        if (ev.type == EV_ABS) switch (ev.code) {
        case ABS_X://X 坐标
            if (!got_curr_x && !got_curr_y) {
                got_curr_x = 1;
                curr_x = ev.value;
            } else {
                got_next_x = 1;
                next_x = ev.value;
            }
            break;
        case ABS_Y://Y 坐标
            if (!got_curr_y) {
                got_curr_y = 1;
                curr_y = ev.value;
            } else {
                got_next_y = 1;
                next_y = ev.value;
            }
            break;
        case ABS_PRESSURE://压力数据
            got_curr_p = 1;
            curr_p = ev.value;
            break;
        }
       //无用信息，继续
        if (!got_curr_x && !got_curr_y && !got_curr_p) continue;
        //时间戳以第一个为准
        if (!got_tstamp) {
```

```
                got_tstamp = 1;
                samp->tv = ev.time;
            }
           //没有读完全一个完整的触摸点信息
            if ( (!got_curr_x || !got_curr_y) && !got_curr_p &&
                !got next x && !got next y ) {
                struct timeval tv = {0, 0};
                fd set fdset;
                FD_ZERO(&fdset);
                FD_SET(ts->fd, &fdset);
                ret = select(ts->fd+1, &fdset, NULL, NULL, &tv);
                    if (ret == 1) continue;
                if (ret == -1) break;
            }
            //已经获取一个完整的触摸点信息
            samp->x = curr_x;
            samp->y = curr y;
            samp->pressure = curr_p;
          printf("RAW---------------------> %d %d %d\n",samp->x,samp->y,
          samp->pressure);
            samp++;
            total++;//采样点计数
            //下一个
            if (got_next_x) curr_x = next_x; else got_curr_x = 0;
            if (got next y) curr y = next y; else got curr y = 0;
            got_next_x = got_next_y = got_tstamp = 0;
        }
        if (ret) ret = -1;
        if (total) ret = total;
#else
```

最后介绍使用 tslib 的步骤。下载 tslib-1.3.tar.tar，解压缩后，运行：

```
#./autogen.sh
```

生成 configure 文件，在该文件最开始加入：

```
CC= arm-linux-gcc
CXX=arm-linux-g++
```

运行：

```
#./configure --host=arm-linux --target=arm-linux --disable-inputapi --prefix=
$PWD/build
#make
#make install
```

生成的目标文件放在当前目录下的 build 目录。将./build/lib 复制到目标板/lib。将./build/share/ts/

plugins 复制到/home/share/ts/plugins。将./build/etc/ts.conf 复制到/home/ts.conf。修改启动脚本，在其中加入以下环境变量的设置：

```
#设置触摸屏类型
export TSLIB TSEVENTTYPE= H3600
#设定控制台设备为 none，否则默认为/dev/tty
export TSLIB_CONSOLEDEVICE=none
#指定帧缓冲设备，默认为/dev/fb0
export TSLIB_FBDEVICE=/dev/fb0
#指定触摸屏设备节点文件
export TSLIB_TSDEVICE=/dev/input/event1
#指定触摸屏校准文件 pintercal 的存放位置
export TSLIB CALIBFILE=/tmp/pointercal
#指定 TSLIB 配置文件的位置
export TSLIB CONFFILE=/home/ts.conf
#指定触摸屏插件所在路径
export TSLIB_PLUGINDIR=/home/share/ts/plugins
```

将 ts_calibrate 复制到目标板，运行后，可以看到屏幕上出现一个十字架，点击该十字架，十字架会跳到另一个角落。共采集 5 次后，计算出最终的结果，放在 TSLIB_CALIBFILE 规定的文件中。

8.6 在 MiniGUI 中加入触摸屏驱动

下面来看一下 MiniGUI 1.3.x 源代码中对 SMDK2410 的触摸屏支持。在初始化的时候打开触摸屏驱动：

```
BOOL Init2410Input (INPUT* input, const char* mdev, const char* mtype)
{
    ts = open ("/dev/ts", O_RDONLY);
    if (ts < 0) {
        fprintf (stderr, "2410: Can not open touch screen!\n");
        return FALSE;
    }
    //设置触摸屏相关的函数接口
    input->update mouse = mouse update;
    input->get_mouse_xy = mouse_getxy;
    input->set_mouse_xy = NULL;
    input->get_mouse_button = mouse_getbutton;
    input->set_mouse_range = NULL;
    input->wait_event = wait_event;
    //初始化坐标
    mousex = 0;
```

```
    mousey = 0;
    ts_event.x = ts_event.y = ts_event.pressure = 0;
    return TRUE;
}
```

在 wait_event 中查询触摸屏的当前位置：

```
tatic int wait_event (int which, fd_set *in, fd_set *out, fd_set *except,struct
timeval *timeout)
{
     …//检查是否有数据，不能采用阻塞式 I/O
     ts event.x=0;
    ts_event.y=0;
    read (ts, &ts_event, sizeof (TS_EVENT));
    if (ts_event.pressure > 0) {
         mousex = ts_event.x;
         mousey = ts_event.y;
    }
    ts event.pressure = ( ts event.pressure > 0 ? 4:0);
    retvalue |= IAL MOUSEEVENT;
}
```

最后在 mouse_getxy 接口中进行校准：

```
static void mouse_getxy(int *x, int* y)
{
    //在此对 mousex、mousey 做坐标校准，最后使用下面语句提交实际坐标
    *x = mousex;
    *y = mousey;
}
```

第9章 块设备驱动

块设备就是支持以块的方式进行读写的设备。块设备与文件系统息息相关。相对 PC 系统而言，在嵌入式系统中，文件系统更强调高效率、稳定性。本章主要介绍块设备驱动的特点、开发方法和文件系统的特点、JFFS2 文件系统的应用。

9.1 Linux 块设备驱动

块设备提供大容量数据的存储功能。它们通常是可移动的单元，而且执行 I/O 操作的速度很慢。块设备的另一个要求是隐藏硬件相关的特性，并提供访问设备的统一接口。例如文件系统不需要关心块设备的底层细节。块设备的第三个特性是当多个请求同时提交给设备时，访问的性能很大程度上取决于请求的顺序。由于块设备有可移动的单元，如果所有的请求是朝同一个方向，性能是最佳的。块设备是作为特殊形式的文件来访问的。它们用主设备号与次设备号来区分。块设备的数据是通过数据块的形式访问的。当文件从设备读取数据时，文件系统中负责读操作的单元会将文件的偏移量转换成块号，然后向该块发送读请求。内核使用电梯算法将 I/O 请求按照顺序放入请求队列。

向内核注册和注销一个块设备可使用如下函数：

```
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
```

下面介绍一些与块设备驱动相关的重要结构：

1. struct gendisk

这个结构保存了一个磁盘的信息。块设备驱动必须分配一个 gendisk，加载分区表，分配请求队列等。

```
struct gendisk {
    int major;                  //主设备号
    int first_minor;
    int minors;                 //最大的次设备号数量，如果设备不能分区，该值为 1
    char disk_name[32];         //主设备名
    struct hd_struct **part;    //分区信息，有 minors 个
    struct block_device_operations *fops;//设备操作
    struct request_queue *queue;//设备管理 I/O 请求
```

```
    void *private_data;
    sector_t capacity;
    int flags;
    char devfs_name[64];
    int number;
    struct device *driverfs_dev;
    struct kobject kobj;
    struct timer_rand_state *random;
    int policy;
    atomic_t sync_io;
    unsigned long stamp, stamp idle;
    int in_flight;
#ifdef  CONFIG_SMP
    struct disk_stats *dkstats;
#else
    struct disk_stats dkstats;
#endif
};
```

Gendisk 的操作函数包括：

```
struct gendisk *alloc_disk(int minors); //分配磁盘
void add_disk(struct gendisk *disk);    //激活磁盘
void del_gendisk(struct gendisk *gp);   //卸载磁盘
static inline void set_capacity(struct gendisk *disk, sector_t size); //设置容量
void blk_queue_hardsect_size(request_queue_t *q, unsigned short size);
                                        //设置扇区大小
```

2. struct hd_struct

这个结构存储了磁盘上的分区信息。

```
struct hd struct {
    sector t start sect;
    sector_t nr_sects;
    struct kobject kobj;
    unsigned reads, read_sectors, writes, write_sectors;
    int policy, partno;
};
```

3. struct bio

用来描述内核以文件系统、虚拟内存子系统或系统调用的形式对块 I/O 设备进行的输入、输出数据的操作。

```
struct bio {
    sector_t        bi_sector;
```

```
    struct bio        *bi_next;        //请求队列链表
    struct block_device *bi_bdev;
    unsigned long       bi_flags;    //状态，命令等等
    unsigned long       bi_rw;
    unsigned short      bi_vcnt;     //bio_vec 的个数
    unsigned short      bi_idx;      //bvl_vec 的当前索引
    unsigned short      bi_phys_segments;
    unsigned short      bi_hw_segments;
    unsigned int        bi_size;     //剩余 I/O 数量
    unsigned int        bi_hw_front_size;
    unsigned int        bi_hw_back_size;
    unsigned int        bi_max_vecs;     //最多可持有的 bvl_vecs 的数量
    struct bio_vec      *bi_io_vec;      //实际的矢量表
    bio_end_io_t        *bi_end_io;
    atomic_t        bi_cnt;
    void            *bi_private;
    bio_destructor_t    *bi_destructor; //销毁器
};
```

4. struct block_device

这个结构代表了内核中的一个块设备。它可以表示整个的磁盘或一个特定的分区。当这个结构代表一个分区时，bd_contains 指向包含这个分区的设备。bd_part 指向设备的分区结构。当这个结构代表一个块设备时，bd_disk 指向设备的 gendisk 结构。

```
struct block_device {
    dev_t           bd_dev;
    struct inode *      bd_inode;        //分区节点
    int         bd_openers;
    struct semaphore    bd_sem;          //打开/关闭锁
    struct semaphore    bd_mount_sem;    //加载互斥锁
    struct list_head    bd_inodes;
    void *          bd_holder;
    int         bd_holders;
    struct block_device *   bd_contains;
    unsigned        bd_block_size;       //分区块大小
    struct hd_struct *  bd_part;
    unsigned        bd_part_count;       //打开次数
    int         bd_invalidated;
    struct gendisk *    bd_disk;
    struct list_head    bd_list;
    struct backing_dev_info *bd_inode_backing_dev_info;
    unsigned long       bd_private;
};
```

5. struct block_device_operations

这个结构是块设备对应的操作接口。你可以在初始化设备驱动时将块设备的操作接口填充到 struct gendisk 的 fops 成员中。

```
struct block device operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*media_changed) (struct gendisk *);
    int (*revalidate_disk) (struct gendisk *);
    struct module *owner;
};
```

6. struct request_queue

这个结构描述了块设备的请求队列。

```
struct request_queue
{
    struct list_head    queue_head;
    struct request      *last_merge;
    elevator_t      elevator;
    //请求队列列表
    struct request list rq;
    request_fn_proc     *request_fn;
    merge_request_fn    *back_merge_fn;
    merge_request_fn    *front_merge_fn;
    merge_requests_fn   *merge_requests_fn;
    make_request_fn     *make_request_fn;
    prep_rq_fn      *prep_rq_fn;
    unplug_fn       *unplug_fn;
    merge_bvec_fn       *merge_bvec_fn;
    activity_fn     *activity_fn;
    //自动卸载状态
    struct timer_list   unplug_timer;
    int         unplug thresh;
    unsigned long       unplug_delay;   //自动卸载延时
    struct work struct  unplug work;
    struct backing dev info backing dev info;
    void            *queuedata;
    void            *activity_data;
    unsigned long       bounce_pfn;
    int         bounce_gfp;
    unsigned long       queue_flags;//各种队列标志
```

```
    //保护队列结构，避免重入
    spinlock_t      *queue_lock;
    //请求的核心结构
    struct kobject kobj;
    //请求的设置
    unsigned long       nr_requests;    //请求的最大数
    unsigned int        nr_congestion_on;
    unsigned int        nr_congestion_off;
    unsigned short      max_sectors;
    unsigned short      max_phys_segments;
    unsigned short      max_hw_segments;
    unsigned short      hardsect_size;
    unsigned int        max_segment_size;
    unsigned long       seg_boundary_mask;
    unsigned int        dma_alignment;
    struct blk_queue_tag    *queue_tags;
    atomic_t        refcnt;
    unsigned int        in_flight;
    //sg 参数设置
    unsigned int        sg_timeout;
    unsigned int        sg_reserved_size;
};
```

请求队列相关的处理函数包括：

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
//创建队列时提供了一个自旋锁
struct request *elv_next_request(request_queue_t *q);
//获得队列中第一个未完成的请求
void end_request(struct request *req, int uptodate);
//请求完成
void blk_queue_hardsect_size(request_queue_t *q, unsigned short size);
//设置扇区尺寸
```

9.2 简单块设备驱动

本节实现一个简单的块设备驱动，它演示了一个最基本的块设备开发方法。这个块设备分配一块内存，并能实现简单的块设备操作。

```
static struct block_device_operations sbd_ops = {
    .owner      = THIS_MODULE,
    .ioctl      = sbd_ioctl
};
static struct request_queue *Queue;
```

```
//设备描述
static struct sbd_device {
    unsigned long size;
    spinlock_t lock;
    u8 *data;
    struct gendisk *gd;
} Device;
```

模块初始化主要是初始化请求队列，注册块设备驱动：

```
static int  init sbd init(void)
{
    Device.size = nsectors*hardsect_size;
    spin_lock_init(&Device.lock);
    //分配内存空间
    Device.data = vmalloc(Device.size);
    if (Device.data == NULL)
     return -ENOMEM;
   //初始化请求队列，设置处理函数为 sbd_request
    Queue = blk init queue(sbd request, &Device.lock);
    if (Queue == NULL)
        goto out;
    blk queue hardsect size(Queue, hardsect size);
   //注册块设备
   major_num = register_blkdev(major_num, "sbd");
    if (major_num <= 0) {
     printk(KERN_WARNING "sbd: unable to get major number\n");
     goto out;
    }
    //填充 Device.gd
    Device.gd = alloc disk(16);
    if (! Device.gd)
     goto out unregister;
    Device.gd->major = major num;
    Device.gd->first minor = 0;
    Device.gd->fops = &sbd_ops;
    Device.gd->private_data = &Device;
    strcpy (Device.gd->disk_name, "sbd0");
    set_capacity(Device.gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
    Device.gd->queue = Queue;
    add_disk(Device.gd);
    return 0;
out unregister:
    unregister blkdev(major num, "sbd");
out:
    vfree(Device.data);
```

```
    return -ENOMEM;
}
```

模块释放：

```
static void _exit sbd_exit(void)
{
    del gendisk(Device.gd);
    put disk(Device.gd);
    unregister blkdev(major num, "sbd");
    blk cleanup queue(Queue);
    vfree(Device.data);
}
```

实现一个简单的 ioctl 命令 HDIO_GETGEO：

```
int sbd_ioctl (struct inode *inode, struct file *filp,unsigned int cmd, unsigned
long arg)
{
     long size;
     struct hd geometry geo;
    //获取块设备的磁盘信息
     switch(cmd) {
         case HDIO GETGEO:
          size = Device.size*(hardsect_size/KERNEL_SECTOR_SIZE);
          geo.cylinders = (size & ~0x3F) >> 6;
          geo.heads = 4;
          geo.sectors = 16;
          geo.start = 4;
          if (copy_to_user((void *) arg, &geo, sizeof(geo)))
              return -EFAULT;
          return 0;
    }
    return -ENOTTY;//未知命令
}
```

I/O 请求（Request）处理实际上就是对 Device.data 进行操作：

```
static void sbd_request(request_queue_t *q)
{
    struct request *req;
    //获取下一个请求
    while ((req = elv next request(q)) != NULL) {
     if (! blk fs request(req)) {
         printk (KERN NOTICE "Skip non-CMD request\n");
         end request(req, 0);
         continue;
```

```
        }
        sbd_transfer(&Device, req->sector, req->current_nr_sectors,
                req->buffer, rq_data_dir(req));
        end_request(req, 1);
       }
}
static void sbd_transfer(struct sbd_device *dev, unsigned long sector,
                                  unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*hardsect_size;
    unsigned long nbytes = nsect*hardsect_size;
    //判断尺寸
    if ((offset + nbytes) > dev->size) {
     printk (KERN_NOTICE "sbd: Beyond-end write (%ld %ld)\n", offset, nbytes);
     return;
    }
    if (write)
      memcpy(dev->data + offset, buffer, nbytes);
    else
      memcpy(buffer, dev->data + offset, nbytes);
}
```

测试结果：

```
[root@(none) tmp]# ls
demo.ko     flashdisk   images      mkdos       mplayer     sdcard      udisk
[root@(none) tmp]# insmod demo.ko
Using demo.ko
devfs_mk_dir: invalid argument.<6> sbd0: unknown partition table
[root@(none) tmp]# mknod /dev/sbd b 253 0
[root@(none) tmp]# ./mkdos -F 16 /dev/sbd
./mkdos 0.4, 27th February 1997 for MS-DOS/FAT/FAT32 FS
70
/dev/sbd
[root@(none) tmp]# df
Filesystem           1k-blocks      Used Available Use% Mounted on
tmpfs                    31128         0     31128   0% /dev/shm
[root@(none) tmp]# mount -t vfat /dev/sbd /tmp/flashdisk
[root@(none) tmp]# df
Filesystem           1k-blocks      Used Available Use% Mounted on
tmpfs                    31128         0     31128   0% /dev/shm
/dev/sbd                   494         2       492   0% /var/tmp/flashdisk
[root@(none) tmp]#
```

9.3 Linux 文件系统

文件系统是指存储设备上的分区和目录结构。存储设备上可以包含一个或多个文件系统。文件系统包括两大类：非日志文件系统和日志文件系统。

非日志文件系统在工作时，不对文件系统的更改进行日志记录。文件系统通过为文件分配文件块的方式把数据存储在磁盘上。每个文件在磁盘上都会占用一个以上的磁盘扇区，文件系统的工作就是维护文件在磁盘上的存放，记录文件占用了的扇区信息。另外扇区的使用情况也要记录在磁盘上。文件系统在读写文件时，首先找到文件使用的扇区号，然后从中读出文件内容。如果要写文件，文件系统首先找到可用扇区，进行数据追加。同时更新文件扇区使用信息。非日志文件系统工作很稳定，但是它存在不少问题。比如如果系统刚将文件的磁盘分区占用信息（meta-data）写入到磁盘分区中，还没有来得及将文件内容写入磁盘，这时意外发生系统断电的情况，结果会造成文件的内容仍然是老内容，而 meta-data 信息是新内容，二者不一致的情况。非日志文件系统的种类很多，Linux 可以支持种类繁多的文件系统，几乎所有的 Linux 发行版都用 ext2 作为默认的文件系统。ext2 文件系统就是一个非日志文件系统。此外，Linux 支持的其他非日志文件系统还有：FAT、VFAT、HPFS（OS/2）、NTFS（Windows NT）和 Sun 的 UFS 等。

日志文件系统则是在非日志文件系统的基础上，加入了文件系统更改的日志记录。日志文件的设计思想是：跟踪记录文件系统的变化，并将变化内容记录入日志。日志文件系统的思想来自大型数据库系统。数据库操作由多个相关的、相互依赖的子操作组成，任何一个子操作的失败都意味着整个操作的无效性，所以，对数据的任何修改都要求回复到操作以前的状态。日志文件系统采用了类似的技术。日志文件系统在磁盘分区中保存有日志记录，写操作首先是对记录文件进行操作，若整个写操作由于某种原因（如系统断电）而中断，系统重启时，会根据日志记录来恢复中断前的写操作。这个过程只需要几秒钟到几分钟。Linux 系统支持的日志文件系统，包括 ext3，XFS，JFS，JFFS2/3 等。Linux 系统中可以混合使用日志文件系统或非日志文件系统。日志增加了文件操作的时间，但是，从文件安全性角度出发，磁盘文件的安全性得到了重大地提高。

ext3 文件系统是直接从 ext2 文件系统发展而来的，目前 ext3 文件系统已经非常稳定可靠。它完全兼容 ext2 文件系统。用户可以平滑地过渡到一个日志功能健全的文件系统中来。这实际上了也是 ext3 日志文件系统初始设计的初衷。ext3 日志文件系统的特点：

1. 高可用性

系统使用了 ext3 文件系统后，即使在非正常关机后，系统也不需要检查文件系统。宕机发生后，恢复 ext3 文件系统的时间只要数十秒钟。

2. 数据的完整性

ext3 文件系统能够极大地提高文件系统的完整性，避免了意外宕机对文件系统的破坏。在保证数据完整性方面，ext3 文件系统有两种模式可供选择。其中之一就是“同时保持文件系统及数据的一致性”模式。采用这种方式，永远不再会看到由于非正常关机而存储在磁盘上的垃圾文件。

3. 文件系统的速度

尽管使用 ext3 文件系统时，有时在存储数据时可能要多次写数据，但是，从总体上看来，ext3 比 ext2 的性能还要好一些。这是因为 ext3 的日志功能对磁盘的驱动器读写头进行了优化。所以，文件系统的读写性能较之 ext2 文件系统来说，性能并没有降低。

4. 数据转换

由 ext2 文件系统转换成 ext3 文件系统非常容易，只要简单地输入两条命令即可完成整个转换过程，用户不用花时间备份、恢复、格式化分区等。用一个 ext3 文件系统提供的小工具 tune2fs，它可以将 ext2 文件系统轻松转换为 ext3 日志文件系统。另外，ext3 文件系统可以不经任何更改，而直接加载成为 ext2 文件系统。

5. 多种日志模式

ext3 有多种日志模式，一种工作模式是对所有的文件数据及 metadata（定义文件系统中数据的数据，即数据的数据）进行日志记录（data=journal 模式）；另一种工作模式则是只对 metadata 记录日志，而不对数据进行日志记录，也即所谓 data=ordered 或者 data=writeback 模式。系统管理人员可以根据系统的实际工作要求，在系统的工作速度与文件数据的一致性之间做出选择。

嵌入式文件系统一般运行在内存、存储器有限的设备上，所以与 PC 上的文件系统不同，它更加关心文件系统的处理效率、稳定性和占用的空间。很多嵌入式文件系统是支持压缩的，并且是只读的。常用的嵌入式文件系统包括 cramfs、romfs、TrueFFS、JFFS2/3、Yaffs/ Yaffs2 等。表 9.1 是几种针对闪存的嵌入式文件系统的比较：

表 9.1　闪存的嵌入式文件系统比较

文件系统	描述	特点
cramfs	每一页被单独压缩，可以随机页访问，其压缩比高达 2∶1，为嵌入式系统节省了大量的 Flash 存储空间。cramfs 文件系统以压缩方式存储，在运行时解压缩	只读，常用于根文件系统 压缩文件系统，所以不支持应用程序以 XIP 方式运行，所有的应用程序要求被复制到 RAM 里去运行
romfs	一种简单的、紧凑的文件系统，不支持动态擦写保存；μClinux 系统通常采用 romfs 文件系统作为根文件系统	只读 它按顺序存放所有的文件，支持应用程序以 XIP 方式运行
TrueFFS	不是真正的文件系统，只提供中间层。需要在上层配合其他文件系统使用。它建立了文件系统的块与闪存中的块的映射	写平衡 垃圾收集 数据一致性
JFFS2/3	建立在 MTD 上的日志文件系统	写平衡 垃圾收集 能提高闪存的利用率 支持数据压缩
Yaffs/ Yaffs2	类 JFFSX，减少了一些功能，效率高。借鉴了日志系统的思想，但不提供日志机能	速度快 占用内存少 只支持 NAND Flash 不支持压缩

9.4 MTD 驱动分析

Linux 中，MTD（Memory Technology Drivers）包含了所有存储器件，比如传统的 ROM、RAM、Flash 和 DOC（Disk On Chip）。为了尽可能避免对不同的存储设备采用不同的工具，使得对这些器件的操作相互兼容，Linux 内核中加入了 MTD 子系统。MTD 子系统提供了一致且统一的接口，让底层的 MTD 芯片驱动程序无缝地与较高层接口组合在一起。JFFS2，cramfs，Yaffs 等文件系统都可以被安装成 MTD 块设备。MTD 驱动也可以为那些支持 CFI 接口的 NOR 型 Flash 提供支持。虽然 MTD 可以建立在 RAM 上，但它是专为基于 Flash 的设备而设计的。MTD 包含特定 Flash 芯片的驱动程序，开发者要选择适合自己系统的 Flash 芯片驱动。Flash 芯片驱动向上层提供读、写、擦除等基本的操作，MTD 对这些操作进行封装后向用户层提供 MTD char 和 MTD block 类型的设备。MTD char 类型的设备包括/dev/mtd0，/dev/mtdl 等，它们提供对 Flash 原始字符的访问。MTD block 类型的设备包括/dev/mtdblock0，/dev/mtdblock1 等，MTD block 设备是将 Flash 模拟成块设备，这样可以在这些模拟的块设备上创建像 Cramfs，JFFS2 等格式的文件系统。

MTD 驱动层也支持在一块 Flash 上建立多个 Flash 分区，每一个分区作为一个 MTD block 设备，可以把系统软件和数据等分配到不同的分区上，同时可以在不同的分区采用不用的文件系统格式。这一点非常重要，正是由于这一点才为嵌入式系统多文件系统的建立提供了灵活性，MTD 文件系统的示意图如图 9-1 所示。

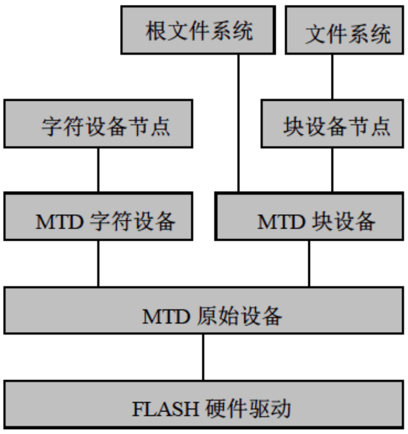


图 9.1　MTD 文件系统

MTD 子系统包含如下几个部分：

（1）Flash 硬件驱动层：硬件驱动层负责在 Init 时驱动 Flash 硬件，Linux MTD 设备的 NOR Flash 芯片驱动遵循 CFI 接口标准，其驱动程序位于 drivers/mtd/chips 子目录下。NAND 型 Flash 的驱动程序则位于/drivers/mtd/nand 子目录下。

（2）MTD 原始设备：原始设备层有两部分组成，一部分是 MTD 原始设备的通用代码，另一部分是各个特定的 Flash 的数据，例如分区。用于描述 MTD 原始设备的数据结构是 mtd_info，这其中定义了大量的关于 MTD 的数据和操作函数。mtd_table（mtdcore.c）则是所有 MTD 原始设备的列表，mtd_part（mtd_part.c）用于表示 MTD 原始设备分区的结构，其中包含了 mtd_info，因为每一个分区都被看成一个 MTD 原始设备加在 mtd_table 中的，mtd_part.mtd_info 中的大部分数据都从该分区的主分区 mtd_part->master 中获得。

（3）MTD 设备层：基于 MTD 原始设备，Linux 系统可以定义出 MTD 的块设备（主设备号 31）和字符设备（设备号 90）。MTD 字符设备的定义在 mtdchar.c 中实现，通过注册一系列 file operation 函数（lseek、open、close、read、write）。MTD 块设备则定义了一个描述 MTD 块设备的结构 mtdblk_dev，并声明了一个名为 mtdblks 的指针数组，数组中的每一个 mtdblk_dev 和 mtd_table 中的每一个 mtd_info 一一对应。

（4）设备节点：通过 mknod 在/dev 子目录下建立 MTD 字符设备节点（主设备号为 90）和 MTD 块设备节点（主设备号为 31），通过访问此设备节点即可访问 MTD 字符设备和块设备。

（5）根文件系统：在 Bootloader 中将 JFFS（或 JFFS2）的文件系统映像 jffs.image（或 jffs2.img）烧到 Flash 的某一个分区中，在/arch/arm/mach-your/arch.c 文件的 your_fixup 函数中将该分区作为根文件系统挂载。

（6）文件系统：内核启动后，通过 mount 命令可以将 Flash 中的其余分区作为文件系统挂载到 mountpoint 上。

一个 MTD 原始设备可以通过 mtd_part 分割成数个 MTD 原始设备注册进 mtd_table，mtd_table 中的每个 MTD 原始设备都可以被注册成一个 MTD 设备，其中字符设备的主设备号为 90，次设备号为 0、2、4、6、…（奇数次设备号为只读设备），块设备的主设备号为 31，次设备号为 0、1、2、3、…。

与其他子系统相似，MTD 子系统与 MTD 工具的开发是独立于内核的。所以内核中的 MTD 代码可能与 MTD 的开发不同步，这是导致一些问题的原因。下面介绍 MTD 子系统最基本的使用方法。在/dev 目录下有 5 种 MTD 设备，如表 9.2 所示：

表 9.2　五种 MTD 设备

/dev entry	Accessible MTD user module	Device type	Major number
mtdN	字符型设备	char	90
mtdrN	字符型设备	char	90
mtdblockN	块设备，只读块设备, JFFS, and JFFS2	block	31
nftlLN	NFTL	block	93
ftlLN	FTL	block	44

MtdN：每个这样的节点代表一个 MTD 设备或一个分区。每个 MTD 分区被认为是一个独立的 MTD 设备。

mtdrN：同 mtdN，区别是此节点代表的设备只读。

mtdblockN：mtdN 节点对应的块设备节点。

nftlLN：代表一个独立的 NFTL 设备。

ftlLN：同 nftlLN。

MTD 子系统包含一系列的工具：

```
#erase device start address number of blocks
//从地址 start_address 开始擦除一定数量的块
#eraseall [options] device
//擦除所有的块
#einfo device
//查看一个 MTD 设备的擦除区域的信息
```

9.5　cramfs 文件系统

cramfs 文件系统是一个压缩的只读文件系统，常用于嵌入式系统作为根文件系统。cramfs 文件系统并不需要一次性地将文件系统中的所有内容都解压缩到内存之中，而只是在系统需要访问某个位置的数据的时侯，马上计算出该数据在 cramfs 中的位置，将其实时地解压缩到内存之中，

然后通过对内存的访问来获取文件系统中需要读取的数据。cramfs 中的解压缩以及解压缩之后的内存中数据存放位置都是由cramfs文件系统本身进行维护的,用户并不需要了解具体的实现过程。在 cramfs 中，文件最大不能超过 16MB。下面是 cramfs 文件系统的常用命令：

```
mkcramfs [-h] [-e edition] [-i file] [-n name] dirname outfile
mkcramfs 用来创建 cramfs 文件系统，各参数的含义如下所示。
-h: 显示帮助信息。
-e edition: 设置生成的文件系统中的版本号。
-i file: 将一个文件映像插入这个文件系统之中（只能在 Linux 2.4.0 以后的内核版本中使用）。
-n name: 设定 cramfs 文件系统的名字。
dirname: 指明需要被压缩的整个目录树。
outfile: 最终输出的文件。
cramfsck [-hv] [-x dir] file
cramfsck 用来进行 cramfs 文件系统的检查，各参数的含义如下所示。
-h: 显示帮助信息。
-x dir: 释放文件到 dir 所指出的目录中。
-v: 输出信息更加详细。
file: 希望测试的目标文件。
```

9.6 NAND 和 NOR Flash

NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发布了 NAND Flash 结构，强调降低每比特的成本，追求更高的性能，并且像磁盘一样可以通过接口轻松升级。但是经过了 10 多年之后，仍然有相当多的硬件工程师分不清 NOR 和 NAND 闪存。

NOR 的特点是芯片内执行（eXecute In Place，XIP），这样应用程序可以直接在 Flash 闪存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1MB～4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于对 Flash 的管理和需要特殊的系统接口。NOR 和 NAND 的比较如表 9.3 所示。

表 9.3 NOR 和 NAND 比较

项目	NOR FLASH	NAND FLASH
读速度	NOR 的读速度比 NAND 稍快一些	
擦除速度	NAND 的擦除速度远比 NOR 快	
写如速度	NAND 的写入速度比 NOR 快很多	
擦除单元	NAND 的擦除单元更小，相应的擦除电路更少	
接口差别	带有 SRAM 接口，有足够的地址引脚来寻址	复杂的 I/O 口来串行地存取数据，各个产品或厂商的方法可能各不相同
容量	小，主要应用在代码存储介质	大，适合于数据存储

续表

项目	NOR FLASH	NAND FLASH
成本	高	低
寿命	最大擦写次数是 10 万次	最大擦写次数是 100 万次
易于使用	非常直接地使用	需要 I/O 接口，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异
软件支持	写入和擦除操作时都需要 MTD	

9.7　在系统中添加 JFFS2 分区

瑞典的 Axis Communications AB 公司开发了 JFFS v1 使用在他们的嵌入式设备中，在 1999 年末基于 GNU GPL 发布出来。最初的发布版本基于 Linux 2.0 内核，后来 RedHat 将它移植到 Linux 2.2 内核，做了大量地测试和 bug fix 的工作使它稳定下来，并且对签约客户提供商业支持。但是在使用的过程中，JFFS v1 设计中的局限被不断地暴露出来。于是在 2001 年初的时候，RedHat 决定实现一个新的闪存文件系统，这就是现在的 JFFS2。下面将详细介绍 JFFS2 设计中主要的思想，关键的数据结构和垃圾收集机制。这将为用户实现一个闪存上的文件系统提供很好的启示。首先，JFFS2 是一个日志结构（log-structured）的文件系统，包含数据和元数据（meta-data）的节点在闪存上顺序地存储。JFFS2 之所以选择日志结构的存储方式，是因为对闪存的更新应该是 out-of-place 的更新方式，而不是对磁盘的 in-place 的更新方式。在闪存上 in-place 更新方式的问题已经在闪存转换层一节描述过了。

JFFS2 克服了 JFFS 中的一些缺点，包括：（1）使用基于哈希（hash）表的日志节点结构，加快操作速度；（2）支持数据压缩；（3）提供“写平衡”支持；（4）支持多种节点，如目录节点、数据节点；（5）提高了对内存的利用率。但 JFFS2 也有不足之处，它的挂载时间过长，磨损平衡是用概率的方法来解决的，具有不确定性。

许多 Linux 的根文件系统采用只读的文件系统，如 romfs、cramfs 文件系统。如果用户需要断电保护数据，需要在系统中加入可写的文件系统。

```
1.  # menu config
```

在文件系统下面，选中 JFFS2 支持，如图 9.2 所示：

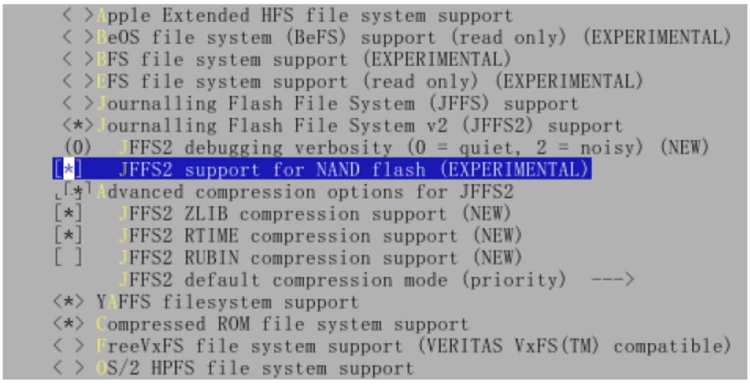


图 9.2　选中 JFFS2 支持

2. 修改/drivers/mtd/maps/pxa27x-flash.c

```
#ifdef CONFIG_ARCH_FS_PXA27X
    {
        .name=   "Bootloader",
        .size=   0x00040000,
        .offset= 0,
        .mask_flags =   MTD_WRITEABLE  //强制只读
    },{
        .name=   "Kernel",
        .size=   0x001C0000,
        .offset= 0x00040000,
    },{
        .name=   "Filesystem",
        .size=   0x01E00000,
        .offset=0x00200000
    },
//改为
static struct mtd_partition pxa27x_partitions[] = {
#ifdef CONFIG ARCH FS PXA27X
    {
        .name=   "Bootloader",
        .size=   0x00040000,
        .offset=0,
        .mask flags =   MTD WRITEABLE       // force read-only
    },{
        .name=   "Kernel",
        .size=   0x001C0000,
        .offset=0x00040000,
    },{
        .name=   "Filesystem",
        .size=   0x01B00000,
        .offset=0x00200000
    },
    {
        .name=   "jffs2",
        .size=   0x00300000,
        .offset=0x01D00000,
    }
```

3. #make zImage

```
/linux-2.6.9-zzm/arch/arm/boot/zImage
```

将内核烧到板子的内核区。安装 JFFS2 文件系统的方法如下所示。

（1）制作 JFFS2 映象可以采用如下命令：

```
./mkfs.jffs2 -r ./jffs/ -o a.img
```

（2）将工具与 a.img 复制到板子/tmp 目录，擦除分区

```
./flash_eraseall  /dev/mtd/3
```

（3）复制映象

```
cp a.img /dev/mtd/3
```

（4）加载文件系统

```
mount -t  jffs2   /dev/mtdblock/3   /mnt/udisk
```

第10章 SD 卡驱动

目前移动设备对存储容量的需求越来越大，SD 卡、CF 卡是非常流行的存储设备。MMC/SD 卡是一类典型的块设备。Linux 内核中已经包含了 MMC 子系统，可以用来支持 MMC/SD 卡。目前很多高端处理器，包括 ARM 都提供了 SD 卡控制器。本章将介绍 SD 卡的协议规范、Linux 下 MMC 驱动的体系结构，最后介绍如何开发 MMC/SD 卡设备驱动。

10.1 SD 卡概述

SD 卡是 Secure Digital Card 卡的简称，直译成汉语就是“安全数字卡”，是由日本松下公司、东芝公司和美国 SANDISK 公司共同开发研制的全新存储卡产品。SD 存储卡是一个完全开放的标准（系统），多用于 MP3、数码摄像机、数码相机、电子图书、AV 器材等，尤其是被广泛应用在超薄数码相机上。SD 卡在外形上与 MultiMedia Card 卡保持一致，大小尺寸比 MMC 卡略厚，容量也大很多，并且兼容 MMC 卡接口规范，SD 卡最大的特点就是通过加密功能，可以保证数据资料的安全保密。它还具备版权保护技术，所采用的版权保护技术是 DVD 中使用的 CPRM 技术（可刻录介质内容保护）。

SD 卡通信基于 9 芯的接口（Clock，Command，Dat[0-3]，Power lines[0-2]），最大的操作频率是 25MHz。。SD 规范包括音频规范、文件系统规范、安全规范、物理层规范等。目前标准 SD 卡已经支持到 2GB 容量，高容量的 SD 卡将支持到 32GB。SD 卡的访问速度分为 4 个等级。高速卡必须支持大于 2MBps 的传输速率。SD 卡协议的各文档之间的结构如图 10.1 所示：

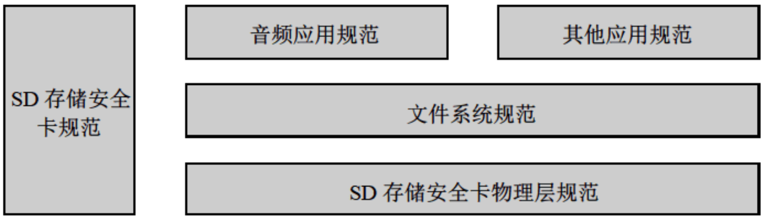


图 10.1 SD 规范的体系结构

SD 卡在结构上使用一主多从的星型拓扑结构。SD 卡系统支持两种通信协议：SD 和 SPI 方式。模式的选择对主机是透明的，由 SD 卡自动检测复位命令的模式，在此后的通信过程中始终使用此种通信方式。SD 卡上电后默认为 SD 模式。SD 模式下共使用 9 个脚位，这包含：时脉、命令、4 条资料线、3 条电源线。CLK：时钟信号；CMD：命令/相应信号；DAT0～DAT3：双向

数据传输信号；VDD，VSS1，VSS2：电源和地信号。SD 的模式管脚如表 10.1 所示。SD 典型接线和外形与接口分别如图 10.2、图 10.3 所示。

表 10.1　SD 模式管脚

管脚	名称	类型	描述	MCI 管脚名
1	CD/DAT[3]	I/O/PP	卡检测/数据线 3	MCDA3/MCDB3
2	CMD	PP	命令线	MCCDA/MCCDB
3	VSS1	S	电源	VSS
4	VDD	S	电源	VDD
5	CLK	I	时钟	MCCK
6	VSS2	S	电源	VSS
7	DAT[0]	I/O/PP	数据线 0	MCDA0/MCDB0
8	DAT[1]	I/O/PP	数据线 1	MCDA1/MCDB1
9	DAT[2]	I/O/PP	数据线 2	MCDA2/MCDB2

备注：I 为输入；O 为输出；PP 为推挽（Push Pull）；S 为电源。

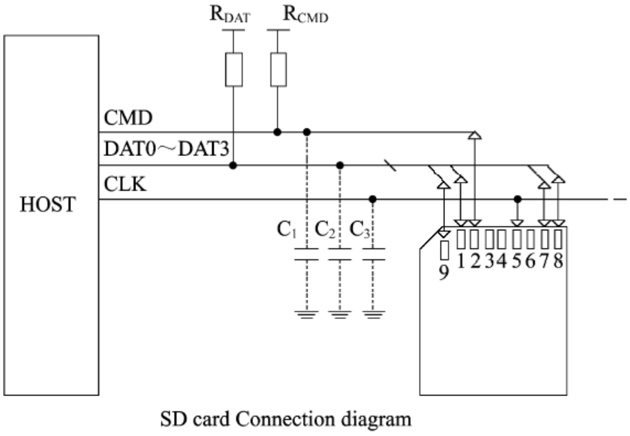


图 10.2　SD 典型接线

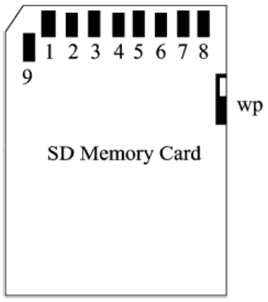


图 10.3　SD 卡的外形和接口

当 SD 卡接收复位命令（CMD0）时如果 $\overline{CS}$ 信号为低电平，则 SD 卡进入 SPI 模式。SPI 模式管脚定义如表 10.2 所示。

表 10.2　SPI 模式管脚

针脚	名称	类型	描述
1	$\overline{CS}$	I	片选（负有效）
2	DI	I	数据输入
3	VSS	S	地
4	VCC	S	供电电压
5	CLK	I	时钟
6	VSS2	S	地
7	DO	O	数据输出
8	RSV	--	
9	RSV	--	

注：S：电源供电，I：输入，O：输出

10.2 SD 卡的通信

SD 总线控制器既可以向特定卡发送命令，又可以向多个连接的 SD 卡发送广播命令。命令字一般包括一个开始位、主机命令标识、命令内容、校验结束位等，总共 48 位。校验采用 16 位的 CCITT 多项式。在 CMD 线上，数据传输的次序是先传输高位后传低位。

SD 卡的命令有 4 种类型，命令格式如表 10.3 所示。

（1）无响应广播命令。

（2）带响应广播命令。各个卡的响应同时进行，这种类型的命令仅用于所用的 CMD 线是分立的，命令和响应会在每根 CMD 线上单独进行。

（3）带地址命令——DAT 线上无数据传输。

（4）带地址命令——DAT 线上有数据传输。

表 10.3　SD 卡命令的格式

位	47	46	45：40	39：8	7：1	0
宽度	1	1	6	32	7	0
值	0	1	X	X	X	1
描述	开始	传输	命令	参数	CRC7	结束

响应字有 5 种根据内容而定的编码方式（R1、R2、R3、R6、R7），长度为 48 位或 136 位，也是通过 CMD 线传送的。

表 10.4 是 R1，R3，R6 编码方式的格式：

表 10.4　SD 卡 48 位的响应格式

位	47	46	45：1	0
宽度	1	1	45	0
值	0	0	X	1
描述	开始	响应	内容	结束

表 10.5 是 R2 编码方式的格式：

表 10.5　SD 卡 136 位的响应格式

位	135	134	133：2	1	0
宽度	1	1	132	1	0
值	0	0	X	X	1
描述	开始	响应	内容	CRC	结束

SD 总线上的通信基于位流的方式，在位流中实现命令和数据，包含起始位和停止位。SD 卡传输数据的单位是块，块数据之后是 CRC 位段。SD 卡传输定义单块和多块的传输。其中，多块传输在快速写入中优于单块传输。在数据传输的过程中，标准的 SD 卡总线使用 DAT0 传输数据，SD 卡宽总线采用 DAT0～DAT4 传输数据。

读块时序如图 10.4 所示：

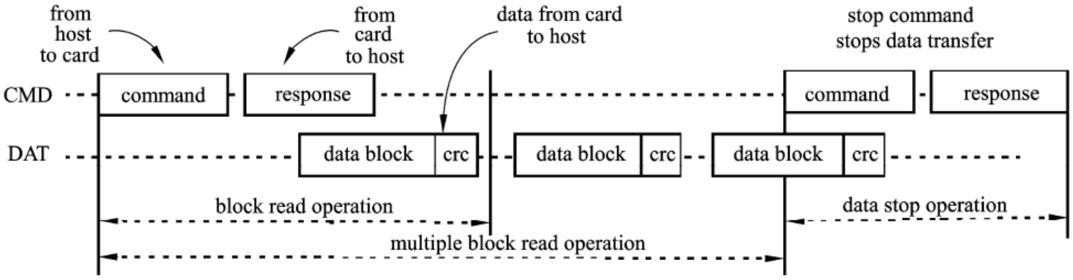


图 10.4　读块时序图

写块时序图如图 10.5 所示：

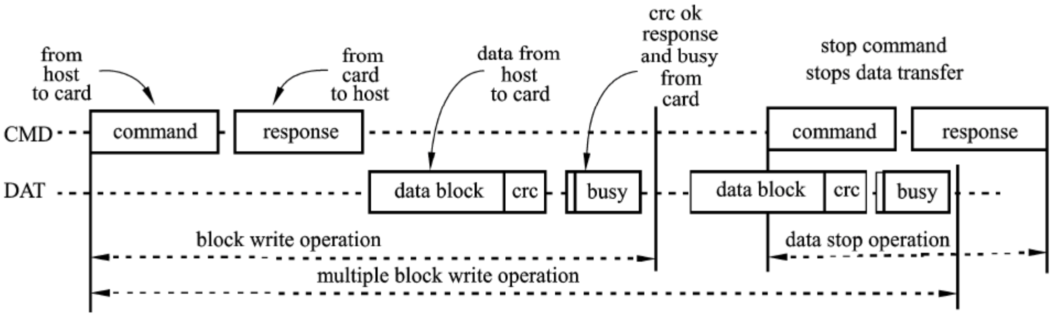


图 10.5　写块时序图

SD 卡的数据有两种打包方式：

（1）普通形式。数据按照 8 位宽度进行组织，LSB 先发送，MSB 后发送，但是具体到每个字节，是 MSB 先发送，LSB 后发送。单数据线和四数据线数据传输形式如图 10.6、图 10.7 所示。

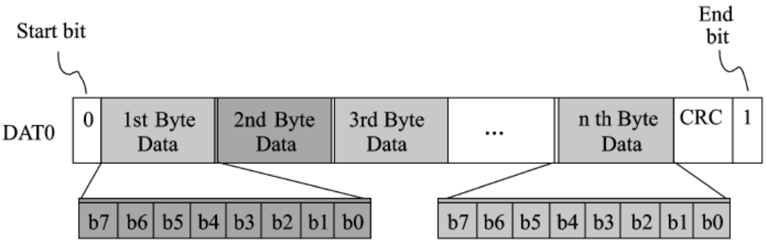


图 10.6　单数据线数据传输普通形式

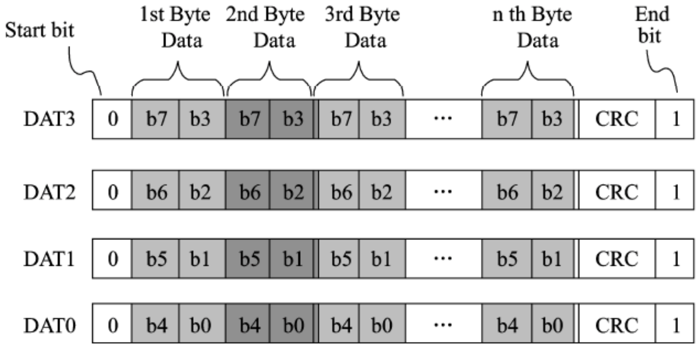


图 10.7　四数据线数据传输普通形式

（2）长数据形式。数据被看作一串二进制数，MSB 位先发送。单数据和四数据线数据传输长数据形式如图 10.8、图 10.9 所示。

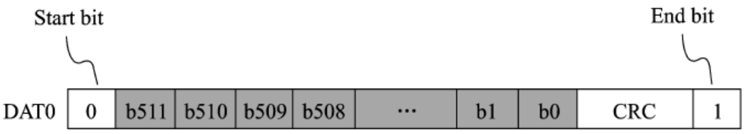


图 10.8　单数据线数据传输长数据形式

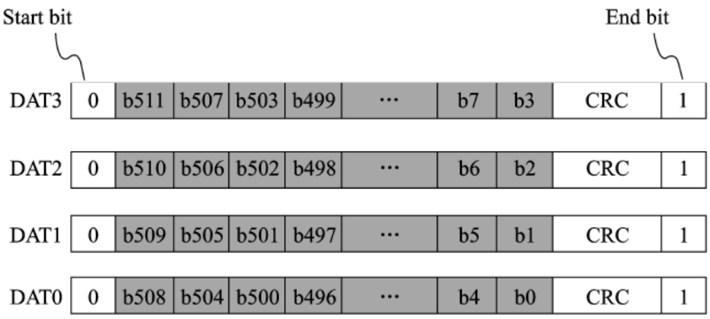


图 10.9　四数据线数据传输长数据形式

10.3　SD 卡寄存器

SD 卡有 OCR、CID、CSD、RCA、DSR、SCR 等 6 个寄存器，它们通过相应的命令字访问。其中 OCR、CID、CSD、SCR 寄存器携带卡的相关信息，RCA 和 DSR 是配置寄存器，保存配置参数。

（1）OCR 寄存器保存了卡的电压特性和供电状态信息，共 32 位，如表 10.6 所示。

表 10.6　OCR 寄存器位定义

位	定义	位	定义
31	上电状态（1：上电过程完成）	18	3.0～3.1
30	能力状态（1：大容量 SD；0：标准容量 SD）	17	2.9～3.0
29：24	保留	16	2.8～2.9
23	3.0～3.1	15	2.7～2.8
22	3.0～3.1	14：8	保留
21	3.0～3.1	7	为低压范围保留
20	3.0～3.1	6：0	保留
19	3.0～3.1		

（2）CID（Card IDentifition）寄存器共 128 位。它包含卡的身份信息，如表 10.7 所示。

表 10.7　CID 寄存器位定义

位	宽度	域	名称
127：120	8	MID	厂商 ID
119：104	16	OID	OEM ID
103：64	40	PNM	产品名
63：56	8	PRV	修订说明
55：24	32	PSN	序列号
23：20	4	--	保留
19：8	12	MDT	
7：1	7	CRC	校验
0	1	--	保留，通常为 1

（3）CSD（Card-Specific Data）寄存器提供了访问卡内容相关的信息，如卡版本、存储速度、支持的命令集、读写数据块大小、擦除的粒度、文件系统的类型等，如图 10.8 所示。

表 10.8　CSD 寄存器位定义

位	宽度	域	名称
127:126	2	CSD_STRUCTURE	CSD 结构标识
125:120	6	-	保留
119:112	8	TAAC	数据读取时间
111:104	8	NSAC	以 CLK 为单位的数据读取时间
103:96	8	TRAN_SPEED	最大数据传输速率
95:84	12	CCC	支持的命令等级
83:80	4	READ_BL_LEN	最大读块尺寸
79	1	READ_BL_PARTIAL	允许部分块读取
78	1	WRITE_BLK_MISALIGN	写块不对齐
77	1	READ_BLK_MISALIGN	读块不对齐
76	1	DSR_IMP	DSR 是否实现
75:70	6	-	保留
69:48	22	C_SIZE	设备容量
47	1	-	保留
46	1	ERASE_BLK_EN	允许擦除单/多个 512 字节块
45:39	7	SECTOR_SIZE	擦除扇区大小
38:32	7	WP_GRP_SIZE	写保护组代大小
31	1	WP_GRP_ENABLE	写保护组允许
30:29	2	-	保留
28:26	3	R2W_FACTOR	写速度因子
25:22	4	WRITE_BL_LEN	最大写块尺寸
21	1	WRITE_BL_PARTIAL	允许部分块的写入
20:16	5	-	保留
15	1	FILE_FORMAT_GRP	文件格式组
14	1	COPY	复制标志
13	1	PERM_WRITE_PROTECT	永久写保护
12	1	TMP_WRITE_PROTECT	暂时写保护
11:10	2	FILE_FORMAT	文件格式

续表

位	宽度	域	名称
9:8	2	-	保留
7:1	7	CRC	校验
0	1	-	保留

（4）RCA 寄存器共 16 位，它存储了卡的地址信息。它是可写的，在卡识别过程中由主机指定。默认的值是 0x0000。此值同样用于设置所有的卡进入 Stand-by 状态（CMD7）。

（5）DSR 寄存器是可选的，共 16 位，用于提高总线的性能。CSD 寄存器的 DSR_IMP 域描述了 DSR 的使用情况，默认值为 0x404，如表 10.9 所示。

表 10.9　DSR_IMP 域

DSR_IMP	DSR_TYPE	DSR_IMP	DSR_TYPE
0	DSR 寄存器没有实现	1	已实现 DSR 寄存器

（6）SCR（SD CARD Configuration Register）共 64 位，包含了特定卡的特性信息，它一般由厂商设定，如表 10.10 所示。

表 10.10　SCR 寄存器位定义

位	宽度	域	名称
63：60	4	SCR_STRUCTURE	SCR 结构标识
59：56	4	SD_SPEC	规范版本
55	1	DATA_STAT_AFTER_ERASE	擦除后的数据状态
54：52	3	SD_SECURITY	安全方面的支持
51：48	4	SD_BUS_WIDTHS	数据线宽支持
47：32	16	--	保留
31：0	32	--	为厂商保留

10.4　Linux 对 SD 卡的支持

Linux 2.6 代码/drivers/mmc 下面的文件就是 MMC/SD 卡的驱动。Linux 的 MMC/SD 卡驱动分为如表 10.11 所示的几个层次：

表 10.11　Linux 的 MMC/SD 卡驱动层次

层次	内容	文件
MMC 协议层 块设备驱动	封装 MMC 协议，实现块设备操作接口	mmc_block.c 调用 mmc_queue.c + mmc_sysfs.c /mmc/mmc_queue.c 实现块设备的读写请求 mmc_sysfs.c 块设备接口
MMC 抽象设备层	Mmc_host/mmc_driver/mmc_card 结构	mmc.c
具体设备层	MMC 控制器接口	用户需要编写的部分，如 pxa27xmci.c

10.4.1 重要数据结构

下面是几个与 MMC/SD 卡驱动相关的重要数据结构。第一个是 MMC 主机的结构：

```
//linux/include/linux/mmc/host.h
struct mmc_host {
    struct device       *parent;
    struct device       class_dev;
    int         index;
    const struct mmc_host_ops *ops;     //操作函数集
    unsigned int        f_min;
    unsigned int        f_max;
    u32         ocr_avail; //可用的 OCR 值
    unsigned long       caps;//主机能力
    //主机专用块
    unsigned int        max_seg_size;   //最大段尺寸
    unsigned short      max_hw_segs;    //最大硬件段个数
    unsigned short      max_phys_segs;  //最大物理段个数
    unsigned short      max_sectors;    //最大扇区数
    unsigned short      unused;
    //私有数据
    struct mmc_ios      ios;            //当前的 I/O 总线状态
    u32         ocr;                    //当前的 OCR 值
    unsigned int        mode;           //当前主机的卡模式
#define MMC_MODE_MMC    0
#define MMC_MODE_SD     1
    struct list_head    cards;          //连接此控制器上的设备
    wait_queue_head_t   wq;             //等待队列
    spinlock_t      lock;               //忙锁
    struct mmc_card     *card_busy;     //与主控制器通信的卡
    struct mmc_card     *card_selected;     //选中的 MMC 卡
    struct delayed_work detect;
    unsigned long       private[0]  cacheline aligned;
};
```

MMC 控制器操作函数：

```
struct mmc_host_ops {
    void    (*request)(struct mmc_host *host, struct mmc_request *req);
    void    (*set_ios)(struct mmc_host *host, struct mmc_ios *ios);
    int (*get_ro)(struct mmc_host *host); //只读保护测试函数
};
```

mmc_ios 结构存储 MMC 卡的状态信息。

```
struct mmc_ios {
    unsigned int    clock;      //时钟
    unsigned short   vdd;       //电源
    unsigned char   bus_mode;   //命令输出模式
    unsigned char   power_mode; //电源供给模式
    unsigned char   bus_width;  //总线宽度
};
```

MMC 请求队列结构：

```
struct mmc_queue {
    struct mmc_card     *card;
    struct task_struct  *thread;
    struct semaphore    thread_sem;
    unsigned int        flags;
    struct request      *req;
    int         (*prep_fn)(struct mmc_queue *, struct request *);
    int         (*issue_fn)(struct mmc_queue *, struct request *);
    void            *data;
    struct request_queue    *queue;
    struct scatterlist  *sg;
};
```

mmc_cid、mmc_csd 是 SD 卡的 CID 寄存器和 CSD 寄存器相关的结构。

```
linux/include/linux/mmc/card.h
struct mmc_cid {
    unsigned int        manfid;
    char            prod_name[8];
    unsigned int        serial;
    unsigned short      oemid;
    unsigned short      year;
    unsigned char       hwrev;
    unsigned char       fwrev;
    unsigned char       month;
};
struct mmc_csd {
    unsigned char       mmca_vsn;
    unsigned short      cmdclass;
    unsigned short      tacc_clks;
    unsigned int        tacc_ns;
    unsigned int        r2w_factor;
    unsigned int        max_dtr;
    unsigned int        read_blkbits;
    unsigned int        write_blkbits;
    unsigned int        capacity;
```

```
    unsigned int        read partial:1,read misalign:1,
                write_partial:1,write_misalign:1;
};
```

mmc_card 代表了一张 MMC/SD 卡。

```
struct mmc_card {
    struct list_head    node;        //在主机设备列表中的节点
    struct mmc_host     *host;       //隶属的主机
    struct device       dev;         //设备结构
    unsigned int        rca;         //相关的卡片地址
    unsigned int        state;       //卡状态
#define MMC_STATE_PRESENT   (1<<0)  //在 sysfs 中存在
#define MMC_STATE_DEAD      (1<<1)  //不活动的设备
#define MMC_STATE_BAD       (1<<2)  //未识别的设备
#define MMC_STATE_SDCARD    (1<<3)  //识别的 SD 卡
#define MMC_STATE_READONLY  (1<<4)  //只读卡
#define MMC_STATE_HIGHSPEED (1<<5)  //高速卡
    u32         raw_cid[4];          //原始卡 CID 信息
    u32         raw_csd[4];          //原始卡 CSD 信息
    u32         raw_scr[2];          //原始卡 SCR 信息
    struct mmc_cid      cid;         //卡身份信息
    struct mmc_csd      csd;         //卡 CSD 信息
    struct mmc_ext_csd  ext_csd;     //mmc v4 扩展 CSD 信息
    struct sd_scr       scr;         //SCR 信息
    struct sd_switch_caps   sw_caps;
};
```

mmc_driver 是 MMC/SD 卡设备驱动结构。

```
struct mmc_driver {
    struct device driver drv;
    int (*probe)(struct mmc_card *);
    void (*remove)(struct mmc_card *);
    int (*suspend)(struct mmc_card *, pm_message_t);
    int (*resume)(struct mmc_card *);
    };
int mmc_register_driver(struct mmc_driver *);
void mmc_unregister_driver(struct mmc_driver *);
```

10.4.2 MMC/SD 卡块设备驱动

下面以 Linux 2.6.20 下 MMC/SD 卡块设备驱动程序（mmc_block.c）为例，分析 MMC/SD 卡的块设备接口。块设备驱动把对 MMC/SD 卡的访问统一到文件系统中，对于应用程序来说，MMC/SD 卡就是一个简单的目录节点。先定义一个 mmc_driver。

```
static struct mmc_driver mmc_driver = {
    .drv        = {
        .name   = "mmcblk",
    },
    .probe      = mmc_blk_probe,
    .remove     = mmc_blk_remove,
    .suspend    = mmc_blk_suspend,
    .resume     = mmc_blk_resume,
};
```

注册 MMC 块设备驱动：

```
static int _init mmc_blk_init(void)
{
    int res = -ENOMEM;
   //注册一个块设备
    res = register_blkdev(mmc_major, "mmc");
    if (res < 0) {
        printk(KERN_WARNING "Unable to get major %d for MMC media: %d\n",
              mmc_major, res);
        goto out;
    }
    if (mmc_major == 0)
        mmc_major = res;
    devfs_mk_dir("mmc");
   //MMC 卡驱动注册，实际就是 driver_register
    return mmc_register_driver(&mmc_driver);
 out:
    return res;
}
```

mmc_blk_probe 是块设备的探测函数，注意它的参数是 mmc_card *card。

```
static int mmc_blk_probe(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int err;
   //检查卡是否支持所需要的命令
    if (!(card->csd.cmdclass & CCC_BLOCK_READ))
        return -ENODEV;
   //分配并填充 mmc_blk_data
    md = mmc_blk_alloc(card);
    if (IS_ERR(md))
        return PTR_ERR(md);
    err = mmc_blk_set_blksize(md, card);
    if (err)
```

```
            goto out;
        printk(KERN_INFO "%s: %s %s %lluKiB %s\n",
            md->disk->disk_name, mmc_card_id(card), mmc_card_name(card),
            (unsigned long long)(get_capacity(md->disk) >> 1),
            md->read_only ? "(ro)" : "");
        mmc_set_drvdata(card, md);
        add_disk(md->disk);
        return 0;
    out:
        mmc blk put(md);
        return err;
}
```

在 mmc_blk_probe 中调用 mmc_blk_alloc，而 mmc_blk_alloc 中又设置了两个重要的函数 md->queue.prep_fn 和 md->queue.issue_fn：

```
static struct mmc_blk_data *mmc_blk_alloc(struct mmc_card *card)
{
    struct mmc_blk_data *md;
    int devidx, ret;
    devidx = find_first_zero_bit(dev_use, MMC_NUM_MINORS);
    if (devidx >= MMC_NUM_MINORS)
        return ERR_PTR(-ENOSPC);
    _set_bit(devidx, dev_use);
    md = kmalloc(sizeof(struct mmc blk data), GFP KERNEL);
    if (!md) {
        ret = -ENOMEM;
        goto out;
    }
    memset(md, 0, sizeof(struct mmc blk data));
    md->read_only = mmc_blk_readonly(card);
    md->block_bits = 9;
    md->disk = alloc_disk(1 << MMC_SHIFT);
    if (md->disk == NULL) {
        ret = -ENOMEM;
        goto err_kfree;
    }
    spin lock init(&md->lock);
    md->usage = 1;
   //初始化队列
    ret = mmc init queue(&md->queue, card, &md->lock);
    if (ret)
        goto err_putdisk;
    md->queue.prep_fn = mmc_blk_prep_rq;        //准备设备请求
    md->queue.issue_fn = mmc_blk_issue_rq;      //发起设备请求
```

```
        md->queue.data = md;
        md->disk->major = major;
        md->disk->first_minor = devidx << MMC_SHIFT;
        md->disk->fops = &mmc_bdops;
        md->disk->private_data = md;
        md->disk->queue = md->queue.queue;
        md->disk->driverfs_dev = &card->dev;
        sprintf(md->disk->disk_name, "mmcblk%d", devidx);
        blk_queue_hardsect_size(md->queue.queue, 1 << md->block_bits);
        set_capacity(md->disk, card->csd.capacity << (card->csd.read_blkbits - 9));
        return md;
    err_putdisk:
        put_disk(md->disk);
    err_kfree:
        kfree(md);
    out:
        return ERR_PTR(ret);
}
```

mmc_blk_issue_rq 函数显示了 MMC/SD 卡本质的读写操作。MMC/SD 协议中的命令用 struct mmc_command 描述。下面是 mmc_blk_issue_rq 的片段。

```
       struct mmc_command cmd;
        u32 readcmd, writecmd;
        memset(&brq, 0, sizeof(struct mmc_blk_request));
        brq.mrq.cmd = &brq.cmd;
        brq.mrq.data = &brq.data;
        brq.cmd.arg = req->sector << 9;
        brq.cmd.flags = MMC_RSP_R1 | MMC_CMD_ADTC;
        brq.data.blksz = 1 << md->block_bits;
        brq.data.blocks = req->nr_sectors >> (md->block_bits - 9);
        brq.stop.opcode = MMC_STOP_TRANSMISSION;
        brq.stop.arg = 0;
        brq.stop.flags = MMC_RSP_R1B | MMC_CMD_AC;
        mmc_set_data_timeout(&brq.data, card, rq_data_dir(req) != READ);
     if (rq_data_dir(req) != READ &&
            !(card->host->caps & MMC_CAP_MULTIWRITE) &&
            !mmc_card_sd(card))
            brq.data.blocks = 1;
        if (brq.data.blocks > 1) {
            brq.data.flags |= MMC_DATA_MULTI;
            brq.mrq.stop = &brq.stop;
            readcmd = MMC_READ_MULTIPLE_BLOCK;//多块读
            writecmd = MMC_WRITE_MULTIPLE_BLOCK; //多块写
        } else {
```

```
        brq.mrq.stop = NULL;
        readcmd = MMC_READ_SINGLE_BLOCK;        //单块读
        writecmd = MMC_WRITE_BLOCK;             //单块写
    }
    if (rq_data_dir(req) == READ) {
        brq.cmd.opcode = readcmd;
        brq.data.flags |= MMC_DATA_READ;
    } else {
        brq.cmd.opcode = writecmd;
        brq.data.flags |= MMC DATA WRITE;
    }
   brq.data.sg = mq->sg;
    brq.data.sg_len = blk_rq_map_sg(req->q, req, brq.data.sg);
    mmc_wait_for_req(card->host, &brq.mrq);
    if (brq.cmd.error) {
        printk(KERN_ERR "%s: error %d sending read/write command\n",
              req->rq_disk->disk_name, brq.cmd.error);
        goto cmd_err;
    }
```

继续跟踪 mmc_wait_for_req：

```
int mmc_wait_for_req(struct mmc_host *host, struct mmc_request *mrq)
{
    DECLARE_COMPLETION_ONSTACK(complete);
    mrq->done_data = &complete;
    mrq->done = mmc_wait_done;
    mmc_start_request(host, mrq);
   //等待完成事件
    wait for completion(&complete);
    return 0;
}
```

再看看 mmc_start_request 函数：

```
Void  mmc_start_request(struct mmc_host *host, struct mmc_request *mrq)
{
    pr_debug("%s: starting CMD%u arg %08x flags %08x\n",
         mmc_hostname(host), mrq->cmd->opcode,
         mrq->cmd->arg, mrq->cmd->flags);
    WARN_ON(host->card_busy == NULL);
    mrq->cmd->error = 0;
    mrq->cmd->mrq = mrq;
    if (mrq->data) {
        mrq->cmd->data = mrq->data;
        mrq->data->error = 0;
```

```
            mrq->data->mrq = mrq;
            if (mrq->stop) {
                mrq->data->stop = mrq->stop;
                mrq->stop->error = 0;
                mrq->stop->mrq = mrq;
            }
        }
        host->ops->request(host, mrq);
    }
```

上面调用的 request 函数就是编写 SD 卡驱动要实现的函数，将在 10.5 小节介绍。

10.4.3 SD 卡扫描

为了不停检测 SD 卡，必须在驱动中调用下面的函数（/drivers/mmc/mmc.c）：

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
{
    struct mmc_host *host;
    host = mmc_alloc_host_sysfs(extra, dev);
    if (host) {
        spin_lock_init(&host->lock);
        init_waitqueue_head(&host->wq);
        INIT_LIST_HEAD(&host->cards);
        INIT_DELAYED_WORK(&host->detect, mmc_rescan);
        host->max_hw_segs = 1;
        host->max_phys_segs = 1;
        host->max_sectors = 1 << (PAGE_CACHE_SHIFT - 9);
        host->max_seg_size = PAGE_CACHE_SIZE;
    }
    return host;
}
```

内核通过 mmc_rescan 不断扫描 MMC/SD 卡：

```
static void mmc_rescan(void *data)
{
    struct mmc_host *host = data;
    struct list_head *l, *n;
    mmc_claim_host(host);
    if (host->ios.power_mode == MMC_POWER_ON)
        mmc_check_cards(host);
    mmc_setup(host);
    if (!list_empty(&host->cards)) {
        host->ios.clock = mmc_calculate_clock(host);
        host->ops->set_ios(host, &host->ios);
```

```
        }
        mmc_release_host(host);
        list_for_each_safe(l, n, &host->cards) {
            struct mmc_card *card = mmc_list_to_card(l);
           //新卡，注册它
            if (!mmc_card_present(card) && !mmc_card_dead(card)) {
                if (mmc_register_card(card))
                    card->state |= MMC_STATE_DEAD;
                else
                    card->state |= MMC_STATE_PRESENT;
            }
           //死卡，删除它
            if (mmc_card_dead(card)) {
                list_del(&card->node);
                mmc_remove_card(card);
            }
        }
       //没有卡存在，关闭电源
        if (list_empty(&host->cards))
            mmc_power_off(host);
    }
```

在 mmc_rescan 中调用了 host->ops->set_ios(host, &host->ios)，那是另一个需要实现的函数。将在 10.5 小节中看到。

10.5 如何开发一个 SD 驱动

MMC/SD 卡的核心部分（包括块设备接口、MMC 命令）已经做好，开发 MMC/SD 卡驱动的工作集中在针对特定的 MMC 控制器进行相应地修改。下面通过分析 Linux 源代码中的 PXA MMCI driver 的驱动来阐述开发具体 MMC/SD 驱动的方法。

```
#define DRIVER_NAME "pxa2xx-mci"
static struct platform_driver pxamci_driver = {
    .probe       = pxamci_probe,
    .remove      = pxamci_remove,
    .suspend     = pxamci_suspend,
    .resume      = pxamci_resume,
    .driver      = {
        .name    = DRIVER_NAME,
    },
};
platform_driver_register(&pxamci_driver);
```

定义一个 MMC/SD 操作函数：

```
static const struct mmc_host_ops pxamci_ops = {
    .request    = pxamci_request,
    .get_ro     = pxamci_get_ro,
    .set_ios    = pxamci_set_ios,
};
```

探测函数中调用了 mmc_alloc_host，启动了 SD 卡扫描检测。

```
static int pxamci_probe(struct platform_device *pdev)
{
    struct mmc_host *mmc;
    struct pxamci_host *host = NULL;
    struct resource *r;
    int ret, irq;
    r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    irq = platform_get_irq(pdev, 0);
    if (!r || irq < 0)
        return -ENXIO;
    r = request_mem_region(r->start, SZ_4K, DRIVER_NAME);
    if (!r)
        return -EBUSY;
    mmc = mmc_alloc_host(sizeof(struct pxamci_host), &pdev->dev);
    if (!mmc) {
        ret = -ENOMEM;
        goto out;
    }
    mmc->ops = &pxamci_ops;
    mmc->f_min = CLOCKRATE_MIN;
    mmc->f_max = CLOCKRATE_MAX;
   //设置最大的物理段数量
    mmc->max_phys_segs = NR_SG;
      //硬件 DMA 支持的最大物理段尺寸为一页
    mmc->max_seg_size = PAGE_SIZE;
    host = mmc_priv(mmc);
    host->mmc = mmc;
    host->dma = -1;
    host->pdata = pdev->dev.platform_data;
    mmc->ocr_avail = host->pdata ?
             host->pdata->ocr_mask :
             MMC_VDD_32_33|MMC_VDD_33_34;
    host->sg_cpu = dma_alloc_coherent(&pdev->dev, PAGE_SIZE, &host->sg_dma,
    GFP_KERNEL);
    if (!host->sg_cpu) {
        ret = -ENOMEM;
        goto out;
    }
```

```
    spin_lock_init(&host->lock);
    host->res = r;
    host->irq = irq;
    host->imask = MMC_I_MASK_ALL;
    host->base = ioremap(r->start, SZ_4K);
    if (!host->base) {
        ret = -ENOMEM;
        goto out;
    }
   //确保主机控制器已经关闭
    pxamci_stop_clock(host);
    writel(0, host->base + MMC_SPI);
    writel(64, host->base + MMC_RESTO);
    writel(host->imask, host->base + MMC_I_MASK);
    host->dma = pxa_request_dma(DRIVER_NAME, DMA_PRIO_LOW,
                   pxamci_dma_irq, host);//返回 DMA 通道号
    if (host->dma < 0) {
        ret = -EBUSY;
        goto out;
    }
    ret = request_irq(host->irq, pxamci_irq, 0, DRIVER_NAME, host);
    if (ret)
        goto out;
    platform_set_drvdata(pdev, mmc);
    if (host->pdata && host->pdata->init)
        host->pdata->init(&pdev->dev, pxamci_detect_irq, mmc);
    mmc_add_host(mmc);
    return 0;
 out:
    if (host) {
        if (host->dma >= 0)
            pxa_free_dma(host->dma);
        if (host->base)
            iounmap(host->base);
        if (host->sg_cpu)
          dma_free_coherent(&pdev->dev, PAGE_SIZE, host->sg_cpu, host->sg_dma);
    }
    if (mmc)
        mmc_free_host(mmc);
    release_resource(r);
    return ret;
}
```

在 linux/arch/arm/mach-pxa/dma.c 中有一个 DMA 通道结构：

```
static struct dma channel {
    char *name;
    void (*irq_handler)(int, void *, struct pt_regs *);
    void *data;
} dma_channels[PXA_DMA_CHANNELS];
```

pxa_request_dma 函数的作用就是建立一个 DMA 通道，并注册该通道的 DMA 中断处理函数 irq_handler。在 DMA 中断处理过程（arch/arm/dma.c）中需要调用 irq_handler：

```
static void pxamci_dma_irq(int dma, void *devid)// irq_handler 函数
{
      printk(KERN_ERR "DMA%d: IRQ???\n", dma);
      DCSR(dma) = DCSR_STARTINTR|DCSR_ENDINTR|DCSR_BUSERR;//结束中断
}
static irqreturn_t dma_irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
int i, dint = DINT;
for (i = 0; i < PXA DMA CHANNELS; i++) {
    if (dint & (1 << i)) {
        struct dma_channel *channel = &dma_channels[i];
        if (channel->name && channel->irq_handler) {
            channel->irq_handler(i, channel->data, regs);
        } else {
            //未注册的 DMA 通道，清除中断，并禁止它
            printk (KERN WARNING "spurious IRQ for DMA channel %d\n", i);
            DCSR(i) = DCSR STARTINTR|DCSR ENDINTR|DCSR BUSERR;
        }
    }
}
    return IRQ HANDLED;
}
static int __init pxa_dma_init (void)
{
     int ret;
    ret = request_irq (IRQ_DMA, dma_irq_handler, 0, "DMA", NULL);
    if (ret)
        printk (KERN CRIT "Wow!  Can't register IRQ for DMA\n");
    return ret;
}
arch_initcall(pxa_dma_init);
```

arch_initcall 是一个宏，它用来添加一个内核启动初始化调用。最终的工作在于编写 pxamci_request、pxamci_get_ro、pxamci_set_ios、pxamci_irq 等函数。pxamci_irq 主要处理命令完成中断。

```
static irqreturn_t pxamci_irq(int irq, void *devid)
{
```

```
    struct pxamci_host *host = devid;
    unsigned int ireg;
    int handled = 0;
    ireg = readl(host->base + MMC_I_REG);
    if (ireg) {
        unsigned stat = readl(host->base + MMC_STAT);
        pr_debug("PXAMCI: irq %08x stat %08x\n", ireg, stat);
        if (ireg & END_CMD_RES)//命令结束
            handled |= pxamci cmd done(host, stat);
        if (ireg & DATA_TRAN_DONE)//数据传输结束
            handled |= pxamci data done(host, stat);
    }
    return IRQ_RETVAL(handled);
}
```

其他的函数读者可以自己分析，从略。

第 11 章 网络设备驱动

网络设备驱动是 Linux 内核中第三大类设备驱动。Linux 一贯在网络通信方面非常强大，它的网络协议子系统非常丰富和稳定。网络设备驱动为网络协议子系统提供了硬件支持。另外，红外通信也可以归入网络子系统。网络设备驱动的核心结构是 sk_buff 结构。本章介绍 DM9000 芯片的驱动开发技术，第 12 章将介绍红外设备驱动开发。

11.1 网络驱动基础

Linux 网络驱动程序的体系结构包括应用层、网络协议层、网络驱动程序和网络设备和网络媒介层。在 Linux 中所有网络设备都抽象为一个接口，这个接口提供了对所有网络设备的操作集合。数据结构 struct net_device 就是网络设备接口。它既包括纯软件网络设备接口，如环路（Loopback），又包括硬件网络设备接口，如以太网卡。Linux 的网络系统主要是基于 BSD UNIX 的 Socket 机制。在网络子系统和驱动程序之间定义有专门的数据结构（sk_buff）进行数据的传递。Linux 网络子系统支持对发送数据和接收数据的缓存，提供流量控制机制，提供对多协议的支持。

网络设备作为 Linux 的三类设备之一，它有其非常特殊的地方。网络接口不存在于 Linux 的文件系统中，而是在核心中用一个 device 数据结构表示。每一个字符设备或块设备在文件系统中都存在一个相应的特殊设备文件来表示该设备，如/dev/hda1、/dev/sda1、/dev/tty1 等。网络设备在做数据包发送和接收时，直接通过接口访问，不需要进行文件的操作；而对字符设备和块设备的访问都需通过文件操作界面。网络接口是在系统初始化时实时生成的，对于核心支持的但不存在的物理网络设备，将不可能有与之相对应的 device 结构。而对于字符设备和块设备，即使该物理设备不存在，在/dev 下也必定有相应的特殊文件与之相对应。

每一个具体的网络接口都应该有一个名字，用来在系统中唯一标识一个网络接口。通常一个名字可表明该接口的类型。Linux 对网络设备命名有以下约定，表 11.1 中 N 为一个非负整数。

表 11.1 网络设备命名

接口名称	说明
ethN	以太网接口，包括 10Mbps 和 100Mbps
trN	令牌环接口
slN	SLIP 网络接口
pppN	PPP 网络接口，包括同步和异步
plipN	PLIP 网络接口，其中 N 与打印端口号相同
tunlN	IPIP 压缩频道网络接口
nrN	NetROM 虚拟设备接口

续表

接口名称	说明
isdnN	ISDN 网络接口
dummyN	空设备
lo	回送网络接口

网络设备驱动最重要的结构是 net_device。网络设备注册和注销函数为：

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

结构 net_device 存储一个网络接口的重要信息，是网络驱动程序的核心。它是系统中网络设备的代表，提供了多个设备方法，供操作系统或协议层调用。它是 IP 层与链路层交流的桥梁。在逻辑上，它可以分割为两个部分：可见部分和隐藏部分。可见部分由外部赋值；隐藏部分的域段仅面向系统内部，它们可以随时被改变。下面将对之进行详细地分析和解剖。

```
struct net_device{
    char      name[IFNAMSIZ];//设备的名称，每个名字表示它的设备类型，同类型的多个设备从 0
    向上编号，如以太网设备的编号为/dev/eth0、/dev/eth1 等
    unsigned long   mem_end;     //共享内存的尾地址
    unsigned long   mem_start;   //共享内存的首地址
    unsigned long   base_addr;   //设备的 I/O 地址
    unsigned int    irq;         //设备的中断号
    unsigned char   if_port;     //记录哪个硬件 I/O 端口正在被接口所用，如 BNC，AUI，TP 等
    unsigned char   dma;         //设备用的 DMA 通道
    unsigned long   state;
    struct net_device   *next;   //指向下一个网络设备，用于维护链表
    int (*init)(struct net_device *dev);     //设备初始化函数，只调用一次
    struct net_device   *next_sched;         //指向下一次调度
    int ifindex;
    int iflink;
    //上面两个为设备标识符
    struct net_device_stats* (*get_stats)(struct net_device *dev);
    struct iw_statistics*   (*get_wireless_stats)(struct net_device *dev);
    //返回无线网络通信的统计信息
    struct iw_handler_def * wireless_handlers;
    struct ethtool_ops *ethtool_ops;         //无线网络扩展成员
----------可见部分结束，以下为隐藏部分----------------------------------
    unsigned long       trans_start;         //最后一次成功发送的时间(jiffies)
    unsigned long       last_rx;             //最后一次成功接收的时间(jiffies)
    unsigned short      flags;                   //该域描述了网络设备的能力和特性
    unsigned short      gflags;
    unsigned short      priv_flags;          //同 flags，但对用户不可见
    unsigned short      unused_alignment_fixer; //32 位对齐字节
    unsigned            mtu;                     //设备的 MTU 值
    unsigned short      type;//接口的硬件类型，描述了与该网络接口绑在一起的媒介类型
```

```
    unsigned short      hard_header_len;    //在被传送的包中 IP 头之前的字节数
    void                *priv;              //该指针指向私有数据
    struct net_device   *master;            //本设备所属设备组的主设备
    unsigned char       broadcast[MAX_ADDR_LEN];    // hw bcast add
    unsigned char       dev_addr[MAX_ADDR_LEN];     // hw address
    unsigned char       addr_len;//硬件地址长度，包括广播地址、物理地址、物理地址的长度。
    struct dev_mc_list  *mc_list;//多播的 MAC 地址
    int mc_count;                 //多播 mc_list 的项目数
    int promiscuity;
    int allmulti;
    int watchdog_timeo;
    struct timer_list   watchdog_timer;//看门狗
    void     *atalk_ptr;          //AppleTalk 连接
    void     *ip_ptr;             //IPv4 相关的数据
    void     *dn_ptr;             //DECnet 相关的数据
    void     *ip6_ptr;            //IPv6 相关的数据
    void     *ec_ptr;             //Econet 相关的数据
    void     *ax25_ptr;           //AX.25 相关的数据
    struct list_head    poll_list;
    int          quota;
    int          weight;
    struct Qdisc        *qdisc;//流量控制结构
    struct Qdisc        *qdisc_sleeping;
    struct Qdisc        *qdisc_ingress;
    struct list_head    qdisc_list;
    unsigned long       tx_queue_len;   //每个队列允许的最大帧
    spinlock_t          ingress_lock;   //入口锁
    spinlock_t          xmit_lock;      //hard_start_xmit 锁
    int                 xmit_lock_owner;
    //当前进入 hard_start_xmit 的 CPU 的 id，如果没有，则为-1
    spinlock_t          queue_lock;     //设备队列锁
    atomic_t            refcnt;         //设备的引用计数
    struct list_head    todo_list;
    struct hlist_node   name_hlist;
    struct hlist_node   index_hlist;
    enum { NETREG_UNINITIALIZED=0,      //未初始化
          NETREG_REGISTERING,           //正在注册
          NETREG_REGISTERED,            //注册完毕
          NETREG_UNREGISTERING,         //正在注销
          NETREG_UNREGISTERED,          //注销完毕
          NETREG_RELEASED,              //已经释放
    } reg_state;                        //设备注册状态字
    int features;                       //设备特性
--------以下为设备函数接口-----------------------------------------------------
    void   (*uninit)(struct net_device *dev);//设备离开网络
```

```
    void (*destructor)(struct net_device *dev);//用户引用结束
    int  (*open)(struct net_device *dev);
    //打开网络接口。每当接口被 ifconfig 激活时，网络接口都要被打开
    int  (*stop)(struct net_device *dev);
    //停止设备
    int  (*hard_start_xmit) (struct sk_buff *skb,struct net_device *dev);
    //硬件开始传输
#define HAVE NETDEV POLL
    int  (*poll) (struct net_device *dev, int *quota);//数据是否就绪
    int  (*hard_header) (struct sk_buff *skb,struct net_device *dev,unsigned short
type,void *daddr,void *saddr,unsigned len);
    //这个函数可根据先前得到的源物理地址和目的物理地址建立硬件头（hardware header）。以太网
    接口的默认函数是 eth_header
    int  (*rebuild_header)(struct sk_buff *skb);
    //在一个包被发送之前重建硬件头。对于以太网设备，若有未知的信息，默认函数将
    //使用 ARP 填写
#define HAVE_MULTICAST
    void  (*set_multicast_list)(struct net_device *dev);
    //设置多点传输的地址链表（*mc_list）
    #define HAVE SET MAC ADDR
    int (*set_mac_address)(struct net_device *dev,void *addr);//改变硬件的物理地址
如果网络接口支持改变它的硬件物理地址，就可用这个操作，许多硬件不支持该功能
#define HAVE PRIVATE IOCTL
    int (*do_ioctl)(struct net_device *dev,struct ifreq *ifr, int cmd);
    //执行依赖接口的 ioctl 命令
#define HAVE_SET_CONFIG
    int (*set_config)(struct net_device *dev,struct ifmap *map);
    //改变接口配置。设备的 I/O 地址和中断号可以通过该函数进行实时修改
#define HAVE_HEADER_CACHE
    int (*hard_header_cache)(struct neighbour *neigh,struct hh_cache *hh);
    void  (*header_cache_update)(struct hh_cache *hh,struct net_device
*dev,unsigned char *  haddr);
#define HAVE CHANGE MTU
    int (*change_mtu)(struct net_device *dev, int new_mtu);
//这个函数负责使接口 MTU 改变后生效。如果当 MTU 改变时驱动程序要做一些特殊的事情，
//就应该写这个函数
#define HAVE_TX_TIMEOUT
    void  (*tx_timeout)  (struct net_device *dev);//发送超时处理
    void  (*vlan_rx_register)(struct net_device *dev,struct vlan_group *grp);
    void  (*vlan_rx_add_vid)(struct net_device *dev,unsigned short vid);
    void  (*vlan_rx_kill_vid)(struct net_device *dev,unsigned short vid);
    int (*hard_header_parse)(struct sk_buff *skb,unsigned char *haddr);
    //解析硬件帧头
    int (*neigh_setup)(struct net_device *dev, struct neigh_parms *);
    int (*accept_fastpath)(struct net_device *, struct dst_entry*);
```

```
#ifdef CONFIG NETPOLL
    int netpoll_rx;
#endif
#ifdef CONFIG_NET_POLL_CONTROLLER
    void (*poll_controller)(struct net_device *dev);
#endif
    struct net_bridge_port  *br_port;//网桥端口
#ifdef CONFIG NET DIVERT
    struct divert_blk   *divert;
#endif // CONFIG_NET_DIVERT
    struct class_device class_dev;//设备类结构
    int padded;//填充字节数量
};
```

对于以太网接口 hard_header_len 为 14，这个值可由 MAC 帧的格式得出。MAC 帧格式=目的地址（6 字节）+源地址（6 字节）+数据长度（2 字节）+数据字节（46～1500 字节）+帧检验序列 FCS。

默认状态下，以太网卡是可广播的，同时，它能够进行组播发送。并把接口的广播地址设置为 FF:FF:FF:FF:FF:FF。相关的标志还有：IFF_ALLMULTI，这个标志告诉驱动程序检索来自网络的所有组播数据包。IFF_PROMISC，这个标志设置接口为混杂模式，使接口接收所有数据包。

```
dev->flags = IFF BROADCAST|IFF MULTICAST;
memset(dev->broadcast,0xFF, ETH_ALEN);
```

11.2 sk_buff

Linux 网络各层之间的数据传送都是通过 sk_buff 进行的。sk_buff 提供一套管理缓冲区的方法，它是 Linux 系统网络高效运行的关键。Struct sk_buffer 是 Linux TCP/IP 协议栈中用于管理数据缓冲的结构。sk_buffer 在数据包的发送和接收中起着重要的作用。为了提高网络处理的性能，应尽量避免数据包的备份。Linux 内核开发者们在设计 sk_buffer 结构的时候，充分考虑到这一点。目前 Linux 协议栈在接收数据的时候，需要复制两次：数据包进入网卡驱动后复制一次，从内核空间递交给用户空间应用时再复制一次。

```
struct sk_buff {
    struct sk_buff      *next;
    struct sk_buff      *prev;
    //本结构是一个双向链表
    struct sk_buff_head *list;
    struct sock         *sk;
    //套接字
    struct timeval      stamp;
    //到达或发送的时间戳
    struct net_device   *dev;
```

```
    struct net_device   *input_dev;
    struct net_device   *real_dev;
    //跟踪使用本结构的设备
    union {
        struct tcphdr   *th;
        struct udphdr   *uh;
        struct icmphdr  *icmph;
        struct igmphdr  *igmph;
        struct iphdr    *ipiph;
        struct ipv6hdr  *ipv6h;
        unsigned char   *raw;
    } h;
    //传输层数据包头
    union {
        struct iphdr    *iph;
        struct ipv6hdr  *ipv6h;
        struct arphdr   *arph;
        unsigned char   *raw;
    } nh;
    //网络层数据包头
    union {
        unsigned char   *raw;
    } mac;
    //链路层包头
    struct  dst_entry   *dst;
    struct  sec_path    *sp;
    char                cb[40];
    //控制缓冲。它可以被各网络层使用。可以将私有变量存放在这里。如果想让它们可以跨层使用，必
    须先调用 skb_clone()
    unsigned int        len,data_len,mac_len,csum;
    //协议数据的长度和校验
    unsigned char       local df,cloned,pkt type,ip summed;
    _u32                priority;//优先级
    unsigned short      protocol,security;
    void                (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_NETFILTER
    unsigned long       nfmark;
    _u32                nfcache;
    struct nf_ct_info   *nfct;
#ifdef CONFIG_NETFILTER_DEBUG
        unsigned int        nf_debug;
#endif
#ifdef CONFIG BRIDGE NETFILTER
    struct nf_bridge_info   *nf_bridge;
#endif
```

```
#endif // CONFIG NETFILTER
#if defined(CONFIG_HIPPI)
    union {
        _u32    ifield;
    } private;
#endif
#ifdef CONFIG_NET_SCHED
      _u32   tc_index; //流量控制系数
#ifdef CONFIG NET CLS ACT
    _u32    tc_verd; //流量控制裁决
    _u32    tc_classid; //流量控制级别
 #endif
#endif
    // These elements must be at the end, see alloc_skb() for details.
    unsigned int        truesize;
    //消耗的内存包括 SKB 本身和 data buffer。
    atomic_t        users;
    unsigned char       *head,*data,*tail,*end;
    //数据缓冲的指针
};
```

对 sk_buff 的控制方法按功能分为两种类型。一种是控制整个 buffer 链的方法，另一种是控制数据缓冲区的方法。sk_buff 组织成双向链表的形式，根据网络应用的特点，对链表的操作主要是删除链表头的元素和添加到链表尾。sk_buff 的控制方法都很短小，以尽量减少系统负荷。

```
struct sk_buff *alloc_skb(unsigned int size, int gfp_mask)
//申请一个 sk_buff 并对它初始化。返回值就是申请到的 sk_buff
static inline struct sk_buff *dev_alloc_skb(unsigned int length)
//类似 alloc_skb，在申请好缓冲区后，保留 16 字节的帧头空间。主要用在 Ethernet 驱动程序
void kfree_skb(struct sk_buff *skb)
//释放一个 sk_buff
struct sk_buff *skb_clone(struct sk_buff *skb, int gfp_mask)
//复制一个 sk_buff，但不复制数据部分
struct sk_buff *skb_copy(const struct sk_buff *skb, int gfp_mask)
//完全复制一个 sk_buff
struct sk_buff *skb_dequeue(struct sk_buff_head *list)
//从一个 sk_buff 链表里取出第一个元素。返回取出的 sk_buff，如果链表空则返回 NULL。这是常用的
//一个操作
void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk)
//在一个 sk_buff 链表头放入一个元素
void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
//在一个 sk_buff 链表尾放入一个元素。这也是常用的一个操作。网络数据的处理主要是对一个先进先出
队列的管理，skb_queue_tail()和 skb_dequeue()完成这个工作
void skb_insert(struct sk_buff *old, struct sk_buff *newsk)
//在链表的某个元素前插入一个元素
```

```
void skb_append(struct sk_buff *old, struct sk_buff *newsk)
//在链表的某个元素后插入一个元素。一些协议（如 TCP）对没按顺序到达的数据进行重组时用到
  skb_insert()和 skb_append()
struct sk_buff *skb_pad(struct sk_buff *skb, int pad)
//确保一个缓冲后面跟随一个 0 值填充内存区域
void skb_split(struct sk_buff *skb, struct sk_buff *skb1, const u32 len)
//将一个 sk_buff 分裂成两个部分，以 len 为界
```

11.3 Linux 网络设备驱动架构

图 11.1 是网络设备工作原理图。Linux 网络设备驱动的主要功能就是网络设备的初始化、网络设备的配置、数据包的收发等。

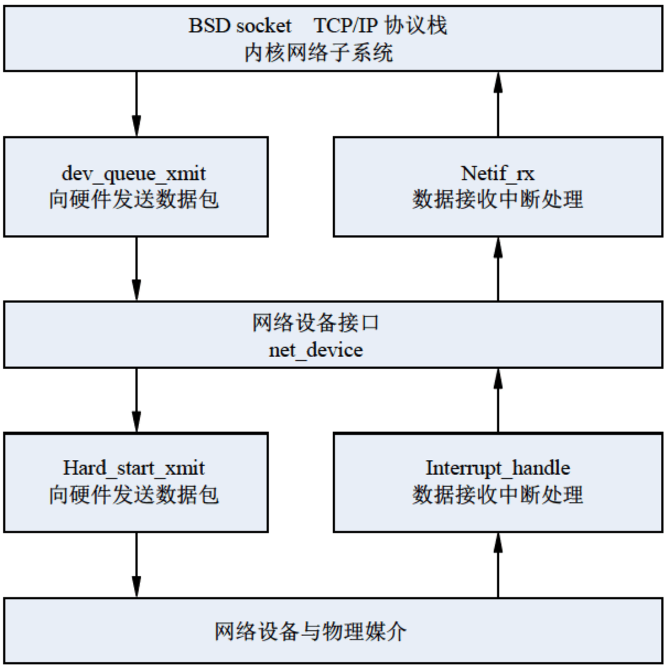


图 11.1 Linux 网络设备工作原理图

操作系统提供了两个接口函数，一个是系统下发数据函数 dev_queue_xmit，它负责调用网络驱动的 hard_start_xmit 接口；一个是驱动上传数据包函数 netif_rx。驱动一般在中断中接收数据，并调用 netif_rx。

```
int netif_rx(struct sk_buff *skb);
int dev_queue_xmit(struct sk_buff *skb);
```

网络设备作为一个对象，提供一些方法供系统访问。正是这些统一的接口函数，屏蔽了硬件的具体细节，让系统对各种网络设备的访问都采用统一的形式，做到硬件无关性。在初始化程序里可以根据硬件的特征检查硬件是否存在，然后决定是否启动这个驱动程序，对其进行配置和初始化。一般需要完成设备 I/O 地址映射、中断申请、DMA 初始化等工作。有些资源是可以和别的

设备共享的，如中断。有些是不能共享的，如 I/O、DMA。接下来要初始化 net_device 结构中的变量，主要包括如下网络设备接口：

1. Open

接口被 ifconfig 激活时，网络接口都要被打开。可以在这个方法中进行资源的申请，硬件的激活等工作。open 这个方法在网络设备驱动程序里是网络设备被激活的时候被调用：

```
#ifconfig eth0 up
```

2. hard_start_xmit

所有的网络设备驱动程序都必须有这个发送方法。在系统调用驱动程序的 hard_start_xmit 时，发送的数据放在一个 sk_buff 结构中。一般的驱动程序把数据传给硬件发出去。有两种特殊的设备，一是 loopback，它把数据组成一个接收数据再回送给系统，二是 dummy 设备，它直接丢弃数据。

如果发送成功，hard_start_xmit 方法释放 sk_buff，返回 0（发送成功）。如果设备暂时无法处理，比如硬件忙，则返回 1。这时如果 dev->tbusy 置为非 0，则系统认为硬件忙，要等到 dev->tbusy 置 0 以后才会再次发送。tbusy 的置 0 任务一般由中断完成。硬件在发送结束后产生中断，这时可以把 tbusy 置 0，然后用 mark_bh()调用通知系统可以再次发送。在发送不成功的情况下，也可以不置 dev->tbusy 为非 0，这样系统会不断尝试重发。如果 hard_start_xmit 发送不成功，则不要释放 sk_buff。传送下来的 sk_buff 中的数据已经包含硬件需要的帧头。所以在发送方法里不需要再填充硬件帧头，数据可以直接提交给硬件发送。

3. stop

stop 方法与 open 方法做相反的工作。可以释放某些资源以减少系统负担。stop 是在设备状态由 up 转为 down 时被调用的：

```
#ifconfig eth0 down
```

4. get_stats

它返回一个 struct net_device_stats 结构，该结构保存了所在网络设备接口的详细流量与错误统计信息：

```
struct net_device_stats
{
    unsigned long   rx_packets; //接收的总包数
    unsigned long   tx_packets; //发送的总包数
    unsigned long   rx_bytes;   //接收的总字节数
    unsigned long   tx_bytes;   //发送的总字节数
    unsigned long   rx_errors;  //收到的错包数量
    unsigned long   tx_errors;  //发送的错包数量
    unsigned long   rx_dropped; //丢弃的接收包数量
```

```
    unsigned long   tx_dropped;          //丢弃的发送包数量
    unsigned long   multicast;           //接收的多播包数
    unsigned long   collisions;
    //详细的接收错误
    unsigned long   rx_length_errors;    //长度错误
    unsigned long   rx_over_errors;      //环行缓冲溢出错误
    unsigned long   rx_crc_errors;       //CRC 校验错误
    unsigned long   rx_frame_errors;     //帧对齐错误
    unsigned long   rx_fifo_errors;      //接收缓冲溢出错误
    unsigned long   rx_missed_errors;    //接收者遗漏错误
    //详细的 tx_errors
    unsigned long   tx_aborted_errors;
    unsigned long   tx_carrier_errors;
    unsigned long   tx_fifo_errors;
    unsigned long   tx_heartbeat_errors;
    unsigned long   tx_window_errors;
    // 为 cslip 等等准备
    unsigned long   rx_compressed;
    unsigned long   tx_compressed;
};
```

5. hard_header

硬件一般都会在上层数据发送之前加上自己的硬件帧头，比如以太网（Ethernet）就有 14 字节的帧头。这个帧头是加在上层 IP、IPX 等数据包的前面的。驱动程序提供一个 hard_header 方法，协议层（IP、IPX、ARP 等）在发送数据之前会调用这段程序。硬件帧头的长度必须放在 dev->hard_header_len，这样协议层会在数据之前保留好硬件帧头的空间。这样 hard_header 程序只要调用 skb_push，然后正确填入硬件帧头就可以了。

6. do_ioctl

此处是设备 ioctl 接口。一般用来实现驱动私有的 ioctl 命令。

7. set_multicast_list

这是一个接口标志，包含了很多值的位掩码。在以太网的默认初始化函数中，该标志被设置为：IFF_BROADCAST|IFF_MULTICAST，表示以太网卡是可广播的，并且是能够进行组播发送的。另外，该标志接口还有一些只读标志，如 IFF_UP，当接口被激活并可以开始传输数据包时，内核设置该标志。而 IFF_PROMISC 被设置或清除时，会调用 set_multicast_list 函数通知板卡上的硬件过滤器。

8. rebuild_header

该函数用来在传输数据包之前，完成 ARP 解析之后，重新建立硬件头。

网络设备驱动程序并不存在一个接收方法。驱动程序收到数据后直接通知操作系统。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 sk_buff（skb），从硬件读

出数据放置到申请好的缓冲区里。接下来填充 sk_buff 中的一些信息。skb->dev = dev，判断收到帧的协议类型，填入 skb->protocol（多协议的支持）。把指针 skb->mac.raw 指向硬件数据然后丢弃硬件帧头（skb_pull）。还要设置 skb->pkt_type，标明第二层（链路层）数据类型。pkt_type 可以是以下类型。

```
PACKET_BROADCAST : 链路层广播。
PACKET_MULTICAST : 链路层组播。
PACKET_SELF : 发给自己的帧。
PACKET_OTHERHOST : 发给别人的帧（监听模式时会有这种帧）。
```


最后驱动程序调用 netif_rx()把数据传送给协议层。netif_rx()将数据放入处理队列后返回，真正的处理是在中断返回以后，这样可以减少中断时间。调用 netif_rx()以后，驱动程序就不能再存取数据缓冲区 skb。

11.4　一个虚拟网络设备驱动

下面来实现一个虚拟的网络设备。这个设备具有硬件收发外的网络设备基本特性。先定义一个结构，包含网络统计信息、sk_buff、自旋锁。

```
struct demo priv
{
    struct net device stats stats;
    struct sk_buff *skb;
    spinlock t lock;
};
```

在模块初始化时注册一个网络设备：

```
struct net_device *demo_devs;
int demo_init_module(void)
{
    int result,ret = -ENOMEM;
    //分配内存，并调用 demo_init
    demo_devs=alloc_netdev(sizeof(struct demo_priv), "fgj%d",demo_init);
    if (demo devs == NULL)
        goto out;
    ret = -ENODEV;
    if ((result = register_netdev(demo_devs)))//注册网络设备
        printk("demo: error %i registering device \"%s\"\n",result, demo_
        devs->name);
    else
        ret = 0;
  out:
    if (ret)
```

```
        demo cleanup();
    return ret;
}
```

在 demo_init 中设置网络操作接口。

```
void demo_init(struct net_device *dev)
{
    struct demo_priv *priv;
    ether setup(dev);
    //设置网络接口函数
    dev->open= demo open;
    dev->stop = demo_release;
    dev->set_config = demo_config;
    dev->hard_start_xmit = demo_tx;
    dev->do_ioctl= demo_ioctl;
    dev->get_stats= demo_stats;
    dev->change_mtu = demo_change_mtu;
    dev->rebuild header= demo rebuild header;
    dev->hard header= demo header;
    dev->tx timeout= demo tx timeout;
    //删除 ARP 支持
    dev->flags|= IFF NOARP;
    dev->features|= NETIF_F_NO_CSUM;
    dev->hard_header_cache = NULL; //禁止高速缓冲
   //私有数据 struct demo_priv 初始化
    priv = netdev_priv(dev);
    memset(priv, 0, sizeof(struct demo_priv));
    spin_lock_init(&priv->lock);
}
```

在打开函数中启动网络处理队列。

```
int demo_open(struct net_device *dev)
{
   memcpy(dev->dev addr, "\0ED000", ETH ALEN);
   netif_start_queue(dev);
   return 0;
}
```

相应地在关闭设备中停止网络处理队列。

```
int demo_release(struct net_device *dev)
{
   netif stop queue(dev);
   return 0;
}
```

demo_tx 实现数据报发送。

```
int demo_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data;
    struct demo_priv *priv = (struct demo_priv *) dev->priv;
    len = skb->len < ETH_ZLEN ? ETH_ZLEN : skb->len;
    data = skb->data;
    //记录时间戳
    dev->trans_start = jiffies;
   //设置 sk_buff 指针
    priv->skb = skb;
    //虚拟硬件发送
    demo_hw_tx(data, len, dev);
    return 0;
}
```

虚拟硬件发送的工作就是更新统计信息，并打印发送的数据报。

```
void demo_hw_tx(char *buf, int len, struct net_device *dev)
{
    struct demo_priv *priv;
    //检查 IP 包的长度，它必须大于 34 字节
    if (len < sizeof(struct ethhdr) + sizeof(struct iphdr)) {
        printk("Bad packet! It's size is less then 34!\n");
        return;
    }
     printk("%s\n",buf);
    //更新统计信息
    priv = (struct demo_priv *) dev->priv;
    priv->stats.tx_packets++;
    priv->stats.rx_bytes += len;
    //释放 sk_buff
    dev_kfree_skb(priv->skb);
}
```

demo_rx 用来接收数据包。

```
void demo_rx(struct net_device *dev, int len, unsigned char *buf)
{
    struct sk_buff *skb;
    struct demo_priv *priv = (struct demo_priv *) dev->priv;
    //组包
    skb = dev_alloc_skb(len+2);
    if (!skb) {
```

```
        printk("demo rx can not allocate more memory to store the packet. drop the
        packet\n");
        priv->stats.rx_dropped++;
        return;
    }
    skb_reserve(skb, 2);
    memcpy(skb_put(skb, len), buf, len);
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);//获得包的协议 ID
    //不需要校验
    skb->ip_summed = CHECKSUM_UNNECESSARY;
    priv->stats.rx_packets++;
    //通知上层
    netif_rx(skb);
    return;
}
```

重建以太包头函数：

```
int demo_header(struct sk_buff *skb,struct net_device *dev,unsigned short type,
                        void *daddr,void *saddr,unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);
    eth->h_proto = htons(type);
    memcpy(eth->h_source,saddr? saddr : dev->dev_addr,dev->addr_len);
    memcpy(eth->h_dest,   daddr? daddr : dev->dev_addr, dev->addr_len);
    return (dev->hard_header_len);
}
```

下面可以编译代码，加载后使用下面的命令测试驱动：

```
~$insmod demo.ko
Using demo.ko
~$ifconfig fgj0 192.168.0.22
~$ping 192.168.0.22
PING 192.168.0.22 (192.168.0.22): 56 data bytes
64 bytes from 192.168.0.22: icmp_seq=0 ttl=64 time=0.6 ms
64 bytes from 192.168.0.22: icmp_seq=1 ttl=64 time=0.4 ms
64 bytes from 192.168.0.22: icmp_seq=2 ttl=64 time=0.3 ms

--- 192.168.0.22 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.3/0.4/0.6 ms
```

11.5　DM9000 网卡芯片

DM9000 是一种集成的廉价快速以太网芯片，它带一个通用处理器接口、一个 10/100 物理接

口和一个 4K 双字节的 SDRAM。DM9000 还提供了一个 MII 接口，用于连接 HPNA 设备或其他支持 MII 的收发器。DM9000 的 PHY 支持 10Base-T 中的 UTP3，4，5，和 100Base-TX 中的 UTP5，它可以自动地配置以发挥最大能力，DM9000 网卡芯片结构图如图 11.2 所示。

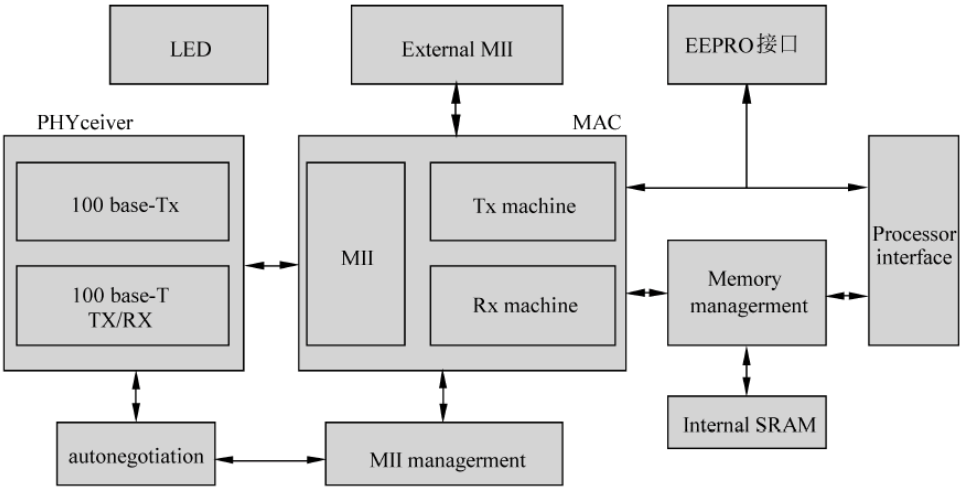


图 11.2　DM9000 结构图

1. 主机接口

主机接口是与 ISA BUS 兼容的模式，用来和系统的主机相连。它有 8 个 I/O 地址，分别是 300H，310H，320H，330H，340H，350H，360H 和 370H。I/O 地址可以通过管脚选择或从 EEPROM 读取。主机接口提供了两种寻址端口。一种是 INDEX 端口，一种是 DATA 端口。当管脚 CMD 为 1，当前访问 DATA 端口，否则访问 INDEX 端口。INDEX 端口的内容就是 DATA 端口的寄存器地址。在访问任何寄存器时，寄存器的地址必须放在 INDEX 端口。

2. DMA 控制

DM9000 提供了 DMA 能力，简化了内部存储器的访问。设定内部存储器的开始地址后，可以通过读写命令启动数据传输。内部存储器的期望地址可以通过读写命令寄存器获得。内部存储器的大小是 16KB。低 3KB 用来作为发送包的数据缓冲，其他 13KB 则留做接受缓冲。因此，在写存储器的操作中，当 IMR（中断掩码寄存器）的第 7 位为 1 时，如果到达结束地址，存储器的地址回到 0。同样，在读存储器过程中，当 IMR 的第 7 位为 1 时，如果到达结束地址，存储器的地址将定位在 0x0C00。

3. 包发送

DM9000 的 TX SRAM 中可以按照顺序同时存储两个发送包，分别为 I 包和 II 包。发送控制寄存器决定 CRC 和填充字节的插入。它们的状态记录在 TX 状态寄存器 I 和 TX 状态寄存器 II 中。复位后发送起始地址是 00H，当前的包是 I 包。第一个包通过 DMA 端口写入 TX SRAM 中，然后将 I 包长度写入 TX 包长度寄存器，将 TX 控制寄存器的 0 位设置为 1，请求发送。DM9000 开始发送 I 包。 I 包发送完成之前， II 包的数据可以被移动到 TX SRAM。 I 包发送完成后，将 II

包长度写入 TX 包长度寄存器，将 TX 控制寄存器的 0 位设置为 1，发送 II 包。如此反复。

4. 包接收

RX SRAM 是一个环形的数据结构。系统复位后 RX SRAM 的开始地址是 0x0C00。每个接收包有一个 4 字节的头，紧接着数据包和 CRC。这个 4 字节的头数据是 01h，状态，字节长度为低位，字节长度为高位。注意每个包的开始地址必须按照操作模式（如 8 位、16 位、32 位）对齐。

5. 重要的寄存器

（1）网络控制寄存器，如表 11.2 所示。

表 11.2　网络控制寄存器位定义

位	名称	默认值	描述
7	EXT_PHY	0，RW	1=选择外部 PHY；0=选择内部 PHY
6	WAKEEN	0，RW	唤醒事件允许：1=允许唤醒功能；0=禁止唤醒功能
5	RESERVED	0，RO	保留
4	FCOL	0，RW	强力碰撞模式，测试用
3	FDX	0，RW	全双工模式
2：1	LBK	00，RW	回环模式 00=正常；01=MAC 内部回环； 10=内部 PHY100M 回环；11=保留
0	RST	0，RW	软复位

（2）网络状态寄存器，如表 11.3 所示。

表 11.3　网络状态寄存器位定义

位	名称	默认值	描述
7	SPEED	0，RO	速率 0=100Mbps；1=10Mbps
6	LINKST	0，RO	连接状态：0=连接失败；1=连接正常，当使用内部 PHY
5	WAKEST	0，RW/C1	唤醒事件状态
4	RESERVED	0，RO	保留
3	TX2END	0，RW/C1	TX 包 2 发送状态
2	TX1END	0，RW/C1	TX 包 1 发送状态
1	RXOV	0，RO	RX FIFO 溢出
0	RESERVED	0，RO	保留

（3）发送控制寄存器，如表 11.4 所示。

表 11.4　发送控制寄存器位定义

位	名称	默认值	描述
7	RESERVED	0，RO	保留
6	TJDIS	0，RW	传输延时允许 1=禁止传输延时计时器；0=允许

续表

位	名称	默认值	描述
5	EXCECM	0，RW	过度碰撞模式控制:0=当过度碰撞计数器超过15中断包发送; 0=仍然尝试发送
4	PAD_DIS2	0，RW	包2的PAD附加禁止
3	CRC_DIS2	0，RW	包2的CRC附加禁止
2	PAD_DIS1	0，RW	包1的PAD附加禁止
1	CRC_DIS1	0，RW	包1的CRC附加禁止
0	TXREQ	0，RW	发送请求

（4）接收控制寄存器，如表11.5所示。

表11.5　接收控制寄存器位定义

位	名称	默认值	描述
7	RESERVED	0，RO	保留
6	WTDIS	0，RW	看门狗计时器禁止 0=允许看门狗计时器；1=禁止看门狗计时器
5	DIS_LONG	0，RW	丢弃长包（包长大于1522字节）
4	DIS_CRC	0，RW	丢弃CRC错误的包
3	ALL	0，RW	通过所有多播包
2	RUNT	0，RW	通过非法的短包（runt packet）
1	PRMSC	0，RW	混杂模式
0	RXEN	0，RW	接收允许

11.6　DM9000网卡驱动程序分析

Linux 2.6中已经带了DM9000的网卡芯片驱动。只需要稍加改动，便可以让DM9000跑起来。下面对修改后的DM9000的代码进行分析，并给出使用方法。先看如何访问DM9000的寄存器，最重要的就是要知道io_addr和io_data。

```
static u8 ior(board_info_t * db, int reg)
{
    writeb(reg, db->io_addr);
    return readb(db->io_data);
}
static void iow(board_info_t * db, int reg, int value)
{
    writeb(reg, db->io_addr);
    writeb(value, db->io_data);
}
```

下面定义设备结构：

```
static struct device driver s3c device drive = {
    .name   = "dm9000",
    .bus    = &platform_bus_type,
    .probe  = dm9000_probe,
    .remove = dm9000_drv_remove,
};
static struct resource s3c_resource_dm9000[] = {
    [0] = {
        .start = 0x20000300 + 0x300,
        .end   = 0x20000300 + 0x300 + 0x03,
        .flags = IORESOURCE MEM
    },
    [1] = {
            .start = 0x20000300 + 0x300 + 0x4,
            .end   = 0x20000300 + 0x300 + 0x4 + 0x7F,
            .flags = IORESOURCE_MEM
   },
    [2] = {
            .start = IRQ EINT14,
            .end   = IRQ EINT14,
            .flags = IORESOURCE IRQ
    }
};
static struct dm9000 plat data s3c device dm9000 platdata = {
    .flags= DM9000_PLATF_16BITONLY
};
struct platform_device s3c_device_dm9000 = {
    .name= "dm9000",
      .id= 0,
      .num_resources= ARRAY_SIZE(s3c_resource_dm9000),
      .resource= s3c resource dm9000,
      .dev= {
      .platform data = &s3c device dm9000 platdata,
    }
};
```

为 DM9000 注册一个平台类设备：

```
static int _init dm9000_init(void)
{
     int ret=0;
    printk(KERN INFO "%s Ethernet Driver\n", CARDNAME);
    platform device register(&s3c device dm9000);
    ret = driver register(&s3c device drive);
      return ret;
}
```

内核自动调用 DM9000_probe:

```
static int dm9000_probe(struct device *dev)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct dm9000_plat_data *pdata = pdev->dev.platform_data;
    struct board_info *db;  //板级信息
    unsigned long base;
    int ret = 0;
    int iosize;
    int i;
    u32 id_val;
    initGPIO();
    //分配网络设备
    ndev = alloc_etherdev(sizeof (struct board_info));
    if (!ndev) {
        printk("%s: could not allocate device.\n", CARDNAME);
        return -ENOMEM;
    }
    SET MODULE OWNER(ndev);
    SET NETDEV DEV(ndev, &pdev->dev);
    PRINTK2("dm9000_probe()");
   //建立电路接口信息结构
    db = (struct board_info *) ndev->priv;
    memset(db, 0, sizeof (*db));
    spin_lock_init(&db->lock);
   //分析 s3c_resource_DM9000 结构变量中的设置,包括中断号、基地址
    if (pdev->num resources < 2) {
        ret = -ENODEV;
        goto out;
    } else if (pdev->num resources == 2) {
        base = pdev->resource[0].start;
        //申请内存资源
        if (!request_mem_region(base, 4, ndev->name)) {
            ret = -EBUSY;
            goto out;
        }
      //芯片基地址
        ndev->base addr = base;
        ndev->irq = pdev->resource[1].start;
      //数据地址与寄存器地址相差 4 字节
        db->io_addr = (void  *)base;
        db->io_data = (void  *)(base + 4);
    } else {
        db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
```

```
        db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
        db->irq_res  = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
            if (db->addr_res == NULL || db->data_res == NULL ||
            db->irq_res == NULL) {
            printk(KERN_ERR PFX "insufficient resources\n");
            ret = -ENOENT;
            goto out;
        }
        i = res_size(db->addr_res);
        db->addr_req = request_mem_region(db->addr_res->start, i,pdev->name);
        if (db->addr_req == NULL) {
            printk(KERN_ERR PFX "cannot claim address reg area\n");
            ret = -EIO;
            goto out;
        }
        db->io_addr = ioremap(db->addr_res->start, i);//地址映射到内核
        if (db->io_addr == NULL) {
            printk(KERN_ERR "failed to ioremap address reg\n");
            ret = -EINVAL;
            goto out;
        }
        iosize = res_size(db->data_res);
        db->data_req = request_mem_region(db->data_res->start, iosize,
            pdev->name);
        if (db->data_req == NULL) {
            printk(KERN_ERR PFX "cannot claim data reg area\n");
            ret = -EIO;
            goto out;
        }
        db->io_data = ioremap(db->data_res->start, iosize);
        if (db->io_data == NULL) {
            printk(KERN_ERR "failed to ioremap data reg\n");
            ret = -EINVAL;
            goto out;
        }
        //填充 net-dev 的成员
        ndev->base_addr = (unsigned long)db->io_addr;
        ndev->irq   = db->irq_res->start;
        //确保至少有一个默认的 I/O 路径
        dm9000_set_io(db, iosize);
    }
    //检查是否要进行硬件设置
    if (pdata != NULL) {
       //设置默认的 I/O 线宽
        if (pdata->flags & DM9000_PLATF_8BITONLY)
```

```
            dm9000_set_io(db, 1);
        if (pdata->flags & DM9000_PLATF_16BITONLY)
            dm9000_set_io(db, 2);
        if (pdata->flags & DM9000_PLATF_32BITONLY)
            dm9000_set_io(db, 4);
        if (pdata->inblk != NULL)
            db->inblk = pdata->inblk;
        if (pdata->outblk != NULL)
            db->outblk = pdata->outblk;
        if (pdata->dumpblk != NULL)
            db->dumpblk = pdata->dumpblk;
    }
    dm9000_reset(db);
    //比较厂商和设备 ID，获取两次，避免出错
    for (i = 0; i < 2; i++) {
        id_val  = ior(db, DM9000_VIDL);
        id_val |= (u32)ior(db, DM9000_VIDH) << 8;
        id_val |= (u32)ior(db, DM9000_PIDL) << 16;
        id_val |= (u32)ior(db, DM9000_PIDH) << 24;
        //ID 核对
        if (id_val == DM9000_ID)
            break;
        printk("%s: read wrong id 0x%08x\n", CARDNAME, id_val);
    }
    if (id_val != DM9000_ID) {
        printk("%s: wrong id: 0x%08x\n", CARDNAME, id_val);
        goto release;
    }
    //找到了 DM9000，填充以太网络设备默认参数
    ether_setup(ndev);
    //设置网络接口
    ndev->open        = &dm9000_open;
    ndev->hard_start_xmit    = &dm9000_start_xmit;
    ndev->tx_timeout        = &dm9000_timeout;
    ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
    ndev->stop        = &dm9000_stop;
    ndev->get_stats       = &dm9000_get_stats;
    ndev->set_multicast_list = &dm9000_hash_table;
#ifdef CONFIG_NET_POLL_CONTROLLER
    ndev->poll_controller    = &dm9000_poll_controller;
#endif
#ifdef DM9000_PROGRAM_EEPROM
    program_eeprom(db);
#endif
    //MII 接口设置
```

```
    db->msg_enable       = NETIF_MSG_LINK;
    db->mii.phy_id_mask  = 0x1f;
    db->mii.reg_num_mask = 0x1f;
    db->mii.force_media  = 0;
    db->mii.full_duplex  = 0;
    db->mii.dev       = ndev;
    db->mii.mdio_read    = dm9000_phy_read;
    db->mii.mdio_write   = dm9000_phy_write;
   //读取 SROM 中的设置
    for (i = 0; i < 64; i++)
        ((u16 *) db->srom)[i] = read_srom_word(db, i);
    //设置节点地址
    for (i = 0; i < 6; i++)
        ndev->dev_addr[i] = db->srom[i];
    //读取 MAC 地址
    if (!is_valid_ether_addr(ndev->dev_addr)) {
        for (i = 0; i < 6; i++)
            ndev->dev_addr[i] = ior(db, i+DM9000_PAR);
    }
    //MAC 地址是否合法
    if (!is_valid_ether_addr(ndev->dev_addr))
        printk("%s: Invalid ethernet MAC address.  Please "
        "set using ifconfig\n", ndev->name);
    dev_set_drvdata(dev, ndev);
   //注册网络设备
    ret = register_netdev(ndev);
    if (ret == 0) {
        printk("%s: dm9000 at %p,%p IRQ %d MAC: ",
            ndev->name,  db->io_addr, db->io_data, ndev->irq);
        for (i = 0; i < 5; i++)
            printk("%02x:", ndev->dev_addr[i]);
        printk("%02x\n", ndev->dev_addr[5]);
    }
    return 0;
release:
out:
    printk("%s: not found (%d).\n", CARDNAME, ret);
    dm9000_release_board(pdev, db);
    kfree(ndev);
    return ret;
}
```

硬件发送函数如下：

```
static int dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
```

```
    board_info_t *db = (board_info_t *) dev->priv;
    PRINTK3("dm9000_start_xmit\n");
    if (db->tx_pkt_cnt > 1)
        return 1;
    //停止队列处理
    netif_stop_queue(dev);
    //禁止所有中断
    iow(db, DM9000_IMR, IMR_PAR);
    //将数据写入 DM9000 发送缓冲
    writeb(DM9000_MWCMD, db->io_addr);
    (db->outblk)(db->io_data, skb->data, skb->len);
    db->stats.tx_bytes += skb->len;
    //发送控制，第一个立即发送，第二个包排队等待
    if (db->tx_pkt_cnt == 0) {
        //第一个包
        db->tx_pkt_cnt++;
        //设置包长度
        iow(db, DM9000_TXPLL, skb->len & 0xff);
        iow(db, DM9000_TXPLH, (skb->len >> 8) & 0xff);
        //发起数据传输
        iow(db, DM9000_TCR, TCR_TXREQ);
        //保存时间戳
        dev->trans_start = jiffies;
    } else {
        //第二个包
        db->tx_pkt_cnt++;
        db->queue_pkt_len = skb->len;
    }
    //释放 SKB
    dev_kfree_skb(skb);
    //重新允许资源检查
    if (db->tx_pkt_cnt == 1)
        netif_wake_queue(dev);
    //允许中断
    iow(db, DM9000_IMR, IMR_PAR | IMR_PTM | IMR_PRM);
    return 0;
}
```

打开操作的主要任务是申请中断。

```
static int  dm9000_open(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;
    PRINTK2("entering dm9000_open\n");
    //申请中断
    if (request_irq(dev->irq, &dm9000_interrupt, SA_INTERRUPT, dev->name, dev))
```

```
        return -EAGAIN;
    //初始化 DM9000
    dm9000_reset(db);
    dm9000_init_dm9000(dev);
    db->dbug_cnt = 0;
    //启动网络连接状态检测
    init_timer(&db->timer);
    db->timer.expires  = DM9000 TIMER WUT;
    db->timer.data     = (unsigned long) dev;
    db->timer.function = &dm9000 timer;
    add timer(&db->timer);
    mii check media(&db->mii, netif msg link(db), 1);
    //启动网络队列处理
    netif_start_queue(dev);
    return 0;
}
```

中断处理包括数据接收中断和数据发送完毕中断：

```
static irqreturn_t dm9000_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct net_device *dev = dev_id;
    board_info_t *db;
    int int_status;
    u8 reg_save;
    PRINTK3("entering %s\n",_FUNCTION_);
    if (!dev) {
        PRINTK1("dm9000_interrupt() without DEVICE arg\n");
        return IRQ_HANDLED;
    }
    // 一个实际的中断发生
    db = (board_info_t *) dev->priv;
    spin lock(&db->lock);
    //保存前一个寄存器地址
    reg save = readb(db->io addr);
    //禁止中断
    iow(db, DM9000_IMR, IMR_PAR);
    //获取中断状态
    int_status = ior(db, DM9000_ISR);   //获取中断状态寄存器
    iow(db, DM9000_ISR, int_status);    //清除中断状态寄存器
    //如果是接收中断
    if (int_status & ISR_PRS)
        dm9000 rx(dev);
    //如果是发送中断
    if (int_status & ISR_PTS)
        dm9000_tx_done(dev, db);
```

```
    //允许中断
    iow(db, DM9000_IMR, IMR_PAR | IMR_PTM | IMR_PRM);
    //恢复前一个寄存器地址
    writeb(reg_save, db->io_addr);
    spin_unlock(&db->lock);
    return IRQ_HANDLED;
}
```

DM9000 接收缓冲中，每个接收包有一个 4 字节的头，包含 0x01、状态和长度：

```
struct dm9000 rxhdr {
    u16 RxStatus;
    u16 RxLen;
} _attribute_((_packed_));
```

DM9000_rx 循环接收数据包，并将数据传递到上层。

```
static void dm9000_rx(struct net_device *dev)
{
    board_info_t *db = (board_info_t *) dev->priv;
    struct dm9000 rxhdr rxhdr;
    struct sk_buff *skb;
    u8 rxbyte, *rdptr;
    int GoodPacket;
    int RxLen;
    //检查包是否就绪
    do {
      //哑读，只读内存数据，不移动指针
        ior(db, DM9000_MRCMDX);
        rxbyte = readb(db->io_data);
        // rxbyte 只能是 0 或 1
        if (rxbyte > DM9000 PKT RDY) {
            printk("status check failed: %d\n", rxbyte);
            iow(db, DM9000_RCR, 0x00); // 停止设备
            iow(db, DM9000_ISR, IMR_PAR);  //停止中断请求
            return;
        }
        //包未就绪
        if (rxbyte != DM9000_PKT_RDY)
            return;
        //包就绪，获取状态和长度
        GoodPacket = TRUE;
        writeb(DM9000 MRCMD, db->io addr);
        //读取包头
        (db->inblk)(db->io data, &rxhdr, sizeof(rxhdr));
        RxLen = rxhdr.RxLen;
```

```
        //包状态检查
        if (RxLen < 0x40) {
            GoodPacket = FALSE;//坏包
            PRINTK1("Bad Packet received (runt)\n");
        }
        //包长大于最大长度
        if (RxLen > DM9000_PKT_MAX) {
            PRINTK1("RST: RX Len:%x\n", RxLen);
        }
       //统计各类错误
        if (rxhdr.RxStatus & 0xbf00) {
            GoodPacket = FALSE;
            if (rxhdr.RxStatus & 0x100) {
                PRINTK1("fifo error\n");
                db->stats.rx_fifo_errors++;
            }
            if (rxhdr.RxStatus & 0x200) {
                PRINTK1("crc error\n");
                db->stats.rx_crc_errors++;
            }
            if (rxhdr.RxStatus & 0x8000) {
                PRINTK1("length error\n");
                db->stats.rx_length_errors++;
            }
        }
        //从 DM9000 中移出数据
        if (GoodPacket&& ((skb = dev_alloc_skb(RxLen + 4)) != NULL)) {
            skb->dev = dev;
            skb_reserve(skb, 2);
            rdptr = (u8 *) skb_put(skb, RxLen - 4);
            //从 DM9000 的 RX  SRAM 中读取数据
            (db->inblk)(db->io_data, rdptr, RxLen);
            db->stats.rx_bytes += RxLen;
            //传递到上层
            skb->protocol = eth_type_trans(skb, dev);
            netif_rx(skb);
            db->stats.rx_packets++;
        } else {
            //抛弃
            (db->dumpblk)(db->io_data, RxLen);
        }
    } while (rxbyte == DM9000_PKT_RDY);
}
```

最后看一下包发送完毕的中断处理函数：

```
static void dm9000_tx_done(struct net_device *dev, board_info_t * db)
{
    //获取发送状态
    int tx_status = ior(db, DM9000_NSR);
    //检查两个发送缓冲的状态是否发送成功一个包
    if (tx_status & (NSR_TX2END | NSR_TX1END))
 {
    //更新统计
        db->tx pkt cnt--;
        db->stats.tx packets++;
      //是否还有包等待发送
        if (db->tx pkt cnt > 0) {
            iow(db, DM9000_TXPLL, db->queue_pkt_len & 0xff);
            iow(db, DM9000_TXPLH, (db->queue_pkt_len >> 8) & 0xff);
            iow(db, DM9000_TCR, TCR_TXREQ);//请求发送
            dev->trans_start = jiffies;
        }
        netif_wake_queue(dev);
    }
}
```

第12章 红外设备驱动

在 Linux 内核中，红外通信是网络子系统的一部分。在应用层，更有 IrSock 红外套接字与一般的网络 Socket 通信相对应。像红外套接字通信一样，也可以自定义一种网络通信协议。本章重点介绍 S3C2410 的红外控制器驱动开发、红外套接字通信等内容。

12.1 红外通信协议规范

各类网络中最具增长潜力的是无线网络，采用无线局域网（WLAN）来拓展现有网络，获得在有效区域内部移动接入网络的能力是目前网络应用研究的热点之一。红外网络通信具有无须申请频率使用权、成本低廉、连接方便、简单易用和结构紧凑等特点，使之与 802.11 协议、蓝牙标准一样，成为三种最流行的短距离无线数据通信的标准。红外数据传输使用的传播介质是红外线。红外线是波长在 750 nm～1 mm 之间的电磁波，是人眼看不到的光线。红外数据传输一般采用红外波段内的近红外线，波长在 0.75 μm～25 μm 之间。红外数据协会成立后，为保证不同厂商的红外产品能获得最佳的通信效果，限定所用红外波长在 850 nm～900 nm。

IrDA 是国际红外数据协会的英文缩写，IrDA 相继制定了很多红外通信协议，有侧重于传输速率方面的，有侧重于低功耗方面的，也有二者兼顾的。IrDA 标准协议如表 12.1 所示。IrDA1.0 协议基于异步收发器 UART，最高通信速率在 115.2Kbps，简称串行红外协议（Serial Infrared，SIR），采用 3/16 ENDEC 编/解码机制。IrDA 1.1 协议提高通信速率到 4Mbps，简称快速红外协议（Fast Infrared，FIR），采用脉冲相位调制（Pulse Position Modulation，4PPM）编译码机制，同时在低速时保留 IrDA 1.0 协议规定。之后，IrDA 又推出了最高通信速率在 16Mbps 的协议，简称特快速红外协议（Very Fast Infrared，VFIR）。

表 12.1 IrDA 标准协议

层次	缩写	名称
基础协议	IrPHY	红外物理层规范
	IrLAP	红外数据链路访问协议
	IrLMP	红外链接管理协议
高层协议	IrCOMM	串并口仿真协议
	TinyTP	传输层协议
	IrLAN	与局域网互连协议
	IrOBEX	对象交换通信协议

IrDA 标准包括三个基本的规范和协议：红外物理层规范（IrPHY）、红外数据链路访问协议（IrLAP）和红外链接管理协议（IrLMP）。物理层规范制定了红外通信硬件设计上的目标和要求，IrLAP 和 IrLMP 为两个软件层，负责对链接进行设置、管理和维护。IrLAP 为 IrDA 设备提供基本链接层连接的协议，在 HDLC 和 SDLC 基础上扩充了一些独特的红外通信特性，提供连接制定、数据转移、流控制等功能，并具有红外线媒质独特属性的附加特点。IrLMP 取决于连接的关系和由 IrLAP 提供的处理特性，它允许多个 Ir 设备连接，并可运行超过一个以上的 IrLAP，解决在搜寻 IrLAP 中的地址冲突，处理在多个设备中的重复地址并产生新的地址，给出连接操作的信息（IAS）。

在 IrLAP 和 IrLMP 的基础上，针对一些特定的红外通信应用领域，IrDA 还陆续发布了一些更高级别的红外协议，如 TinyTP、IrOBEX、IrCOMM、IrLAN、IrTran-P 等等，其中 Tiny TP 是传输层通信协议（IrDA Transport Protocols:Tiny TP），负责管理不同 IrDA 装置之间的虚拟信道（virtual channels），执行调试、将数据分割（segment）成为封包、从封包中重组还原数据。Tiny TP 执行的工作类似 TCP。IrOBEX 是一种对象交换通信协议，它定义了 PUT 和 GET 命令，可以在两台 IrDA 装置之间存取二进制（binary）数据。它位于 Tiny TP 上方，定义了对象交换时，封包所必需的内容，以利于 IrDA 装置通信时能彼此辨识。

12.2 S3C2410X 红外接口

S3C2410X 处理器提供了三个独立的异步串口，它们能够工作在中断模式或 DMA 模式。每个串口通道包含两个 16 字节的收发 FIFO。这三个串口的波特率是可调的，并且都可以作为红外收发器使用，支持 IrDA 1.0 协议。基本的红外数据包括起始位、数据位和结束位。红外发送包中使用 3/16 位脉冲标识数据位 0，接收器在接收到 3/16 位脉冲认为收到 0。图 12.1 和 12.2 显示了红外传输帧的时序图。

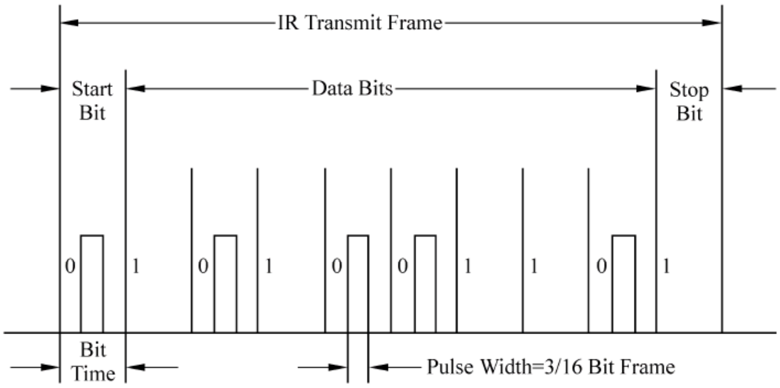


图 12.1 S3C2410X 红外发送帧

S3C2410X 的串口还提供了一种测试模式。这种模式结构上是将 RXD 和 TXD 连接在一起，它使得处理器可以检测内部的数据发送。下面来看看 S3C2410X 中与串口相关的几个重要的寄存器。

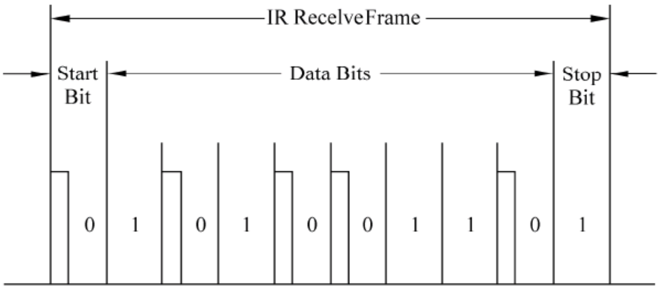


图 12.2　S3C2410X 红外接收帧

（1）串口帧格式寄存器（ULCON），如表 12.2 所示。

表 12.2　串口帧格式寄存器位定义

ULCONn	位	描述	初始值
Reserved	[7]	保留	0
Infrared-Red Mode	[6]	0=正常模式；1=红外收发模式	0
Parity Mode	[5:3]	0XX=无校验；100=奇校验； 101=偶校验；110=强制 1 校验； 111=强制 0 校验	000
Number of Stop bit	[2]	0=每帧一个停止位； 1=每帧两个停止位	0
Word Length	[1:0]	规定每帧的数据位数： 00=5 位；01=6 位；10=7 位；11=8 位	00

（2）串口控制寄存器（UCON），如表 12.3 所示。

表 12.3　串口控制寄存器位定义

UCONn	位	描述	初始值
Clock Selection	[10]	0=PCLK:UBRDIVn=(int)(PCLK/bps*16))–1 1=UCLK: UBRDIVn=(int)(UCLK/bps*16))–1	0
Tx Interrupt Type	[9]	发送中断请求类型：0=脉冲；1=电平	0
Rx Interrupt Type	[8]	接收中断请求类型：0=脉冲；1=电平	0
Rx TimeOut Enable	[7]	Rx 超时中断允许：0=禁止；1=允许	0
Rx Error Status Interrupt Enable	[6]	错误状态中断允许： 0=禁止；1=允许	0
Loopback Mode	[5]	回环模式选择：0=正常模式；1=回环模式	0
Send Break Signal	[4]	0=正常传输；1=发送中断信号	0
Transmit Mode	[3:2]	数据发送方式选择： 00=禁止；01=中断或轮询； 10=DMA0，DMA3；11=DMA1	00
Receive Mode	[1:0]	数据接收方式选择： 00=禁止；01=中断或轮询； 10=DMA0，DMA3；11=DMA1	00

（3）串口收发状态寄存器（UTRSTAT），如表 12.4 所示。

表 12.4　串口收发状态寄存器位定义

UTRSTAT n	位	描述	初始值
Transmitter empty	[2]	当发送缓冲寄存器没有有效数据或发送移位寄存器为空时，自动清零。 0=发送缓冲不为空；1=发送缓冲为空	1
Transmit buffer empty	[1]	当发送缓冲寄存器为空时，自动清零。 0=发送缓冲寄存器不为空；1=为空	1
Receive buffer data ready	[0]	0=接收缓冲寄存器为空； 1=接收缓冲寄存器有数据	0

（4）串口错误状态寄存器（UERSTAT），如表 12.5 所示。

表 12.5　串口错误状态寄存器位定义

UERSTAT n	Bit	描述	初始值
Break Detect	[3]	0=没有检测到中断包； 1=接收到中断包	0
Frame Error	[2]	0=没有收到帧错误信息； 1=收到帧错误信息	1
Parity Error	[1]	0=没有校验错误；1=出现校验错误；	
Overrun Error	[0]	0=没有溢出错误；1=溢出错误	0

（5）串口发送缓冲寄存器（UTXH），如表 12.6 所示。

表 12.6　串口发送缓冲寄存器位定义

UTXH n	位	描述	初始值
TXDATAn	[7:0]	发送的数据	--

（6）串口接收缓冲寄存器（URXH），如表 12.7 所示。

表 12.7　串口接收缓冲寄存器位定义

UTXH n	位	描述	初始值
RXDATAn	[7:0]	接收的数据	--

（7）串口波特率分频寄存器（UBRDIV），如表 12.8 所示。

表 12.8　串口波特率分频寄存器位定义

UBRDIV n	位	描述	初始值
UBRDIV	[15:0]	波特率分频值	--

12.3　S3C2410X 红外设备驱动

下面来实现 S3C2410X 上的红外通信设备驱动，以串口 2 为例。电路设计参考 YL2410 开发

板。首先定义几个访问函数：

```
#define portaddr(reg)  ((void *)((char*)S3C2410_VA_UART2 + (reg)))
#define rd_regb(reg)  (_raw_readb(portaddr(reg)))
#define rd regl(reg)  ( raw readl(portaddr(reg)))
#define wr regl(val,reg) \
  do {  raw writel(val,portaddr(reg)); } while(0)
#define wr regb(val,reg) \
  do { _raw_writeb(val, portaddr(reg)); } while(0)
```

初始化串口 2 的寄存器：

```
void initInfraredPort(void)
{
    wr_regl((int)(PCLK/16/19200)-1,S3C2410_UBRDIV);
    writel((readl(S3C2410 GPBDAT)&0x7fd),S3C2410 GPBDAT);
    wr regl((1<<6)|(0<<4)|(1<<2)|(1<<1)|0,S3C2410 UFCON);
    wr regl((0<<10)|(1<<9)|(1<<8)|(0<<7)|(1<<6)|(0<<5)|(0<<4)|(0<<2)|(0),
    S3C2410 UCON);
    wr regl(((1<<6)|(0<<3)|(0<<2)|(3)),S3C2410 ULCON);
    wr regl((0<<10)|(1<<9)|(1<<8)|(0<<7)|(1<<6)|(1<<5)|(0<<4)|(1<<2)|(1),
    S3C2410_UCON);
}
```

在模块初始化的过程中，注册字符设备，申请中断：

```
if (request irq(IRQ S3CUART RX2,&s3c2410infrared interrupt, SA INTERRUPT, "
s3c2410infrared", NULL)) {
    printk("request infrared irq failed!\n");
    return -1;
}
init_waitqueue_head(&DEMO_devices->wq);
```

红外接收数据中断处理如下：

```
static irqreturn_t s3c2410infrared_interrupt(int irq, void *dummy, struct pt_regs *fp)
{
    int a;
    disable_irq(IRQ_UART2);
    disable irq(IRQ S3CUART RX2);
    disable irq(IRQ S3CUART ERR2);
    //读中断来源
    a=readl(S3C2410 SUBSRCPND);
    if(a&(1<<6))
    {
        a=rd_regl(S3C2410_UTRSTAT);//读收发状态寄存器
        if(a|1)
```

```
            {
                DEMO_devices->infraredbuf=rd_regb(S3C2410_URXH);//获取数据
                flag = 1;
                wake_up_interruptible(&(DEMO_devices->wq));
                //printk("rd_regl:%d\n",DEMO_devices->infraredbuf);
            }
        }
        else
        {
            a=rd_regl(S3C2410_UERSTAT);
        }
        writel(readl(S3C2410_SRCPND)&(~(1<<15)),S3C2410_SRCPND);
        writel(readl(S3C2410_SUBSRCPND)&(~(1<<6)|(1<<8)),S3C2410_SUBSRCPND);
        enable_irq(IRQ_UART2);
        enable_irq(IRQ_S3CUART_RX2);
        enable_irq(IRQ_S3CUART_ERR2);
        return IRQ_HANDLED;
}
```

数据发送的过程相当简单，就是向 UTXH 填充数据。

```
ssize_t DEMO_write(struct file *filp, const char _user *buf, size_t count,loff_t
*f_pos)
{
    unsigned char buff;
    copy_from_user(&buff, buf,1);
    wr_regb(buff,S3C2410_UTXH);
    return 1;
}
```

测试程序如下：

```
int infrared_fd;
unsigned char infrared_value;
unsigned char wbuff=0;
infrared_fd = open("/dev/irda", O_RDWR);
if (infrared_fd < 0) {
  perror("open device /dev/irda");
  exit(1);
}
while(1)
{
        printf("begin to write /dev/irda:%d\n",wbuff);
        write(infrared_fd,&wbuff,1);
        wbuff++;
        int ret = read(infrared_fd, &infrared_value,1);
```

```
        if (ret !=1)
        {
            if (errno != EAGAIN)
            perror("read infrared_fd\n");
            continue;
        }
     else
        {
           printf("get infrared_value: %d\n", infrared_value);
        }
     sleep(1);
  }
  close(infrared_fd);
```

测试结果：

[root@(none) tmp]# insmod demo.ko

Using demo.ko

[root@(none) tmp]# mknod /dev/irda c 224 0

[root@(none) tmp]# ./read

begin to write /dev/irda:0

get infrared_value: 0

begin to write /dev/irda:1

get infrared_value: 1

begin to write /dev/irda:2

get infrared_value: 2

12.4　Linux 对红外网络通信的支持

IrDA 不是单纯的串口物理通信规范，而是一种网络传输控制标准。在 Linux 操作系统下，红外通信是作为一类特殊的网络设备来支持的。在 linux/net/irda 中有红外设备的代码。可以用下面的函数初始化一个红外设备：

```
struct net_device *alloc_irdadev(int sizeof_priv)
{
    return alloc_netdev(sizeof_priv, "irda%d", irda_device_setup);
}
```

在 irda_device_setup 函数中定义了一些网络参数：

```
void irda_device_setup(struct net_device *dev)
{
   dev->hard_header_len = 0;
```

```
    dev->addr len = 0;
    dev->type   = ARPHRD_IRDA;
    dev->tx_queue_len= 8; // Window size + 1 s-frame
     memset(dev->broadcast, 0xff, 4);
     dev->mtu = 2048;
     dev->flags = IFF_NOARP;
}
```


通过分析内核中的 pxaficp_ir.c 来了解红外通信设备驱动的开发方法。首先应该注册红外设备，设置一些必要的网络接口函数：

```
static int pxa_irda_probe(struct platform_device *pdev)
{
    struct net_device *dev;
    struct pxa_irda *si;
    unsigned int baudrate mask;
    int err;
    dev = alloc irdadev(sizeof(struct pxa irda));
    if (!dev)
        goto err_mem_3;
    …
    si = netdev_priv(dev);
    si->dev = &pdev->dev;
    si->pdata = pdev->dev.platform_data;
    dev->hard start xmit    = pxa irda hard xmit;
    dev->open               = pxa irda start;
    dev->stop               = pxa irda stop;
    dev->do ioctl           = pxa irda ioctl;
    dev->get stats          = pxa irda stats;
    err = register netdev(dev);
}
```

在打开函数中申请红外中断，并启动队列。

```
static int pxa_irda_start(struct net_device *dev)
{
    struct pxa_irda *si = netdev_priv(dev);
    int err;
    si->speed = 9600;
    err = request_irq(IRQ_STUART, pxa_irda_sir_irq, 0, dev->name, dev);//标准红外
    if (err)
        goto err_irq1;
    err = request_irq(IRQ_ICP, pxa_irda_fir_irq, 0, dev->name, dev);//快速红外
    if (err)
        goto err_irq2;
  ...
```

```
    //允许中断
    enable_irq(IRQ_STUART);
    enable_irq(IRQ_ICP);
    netif_start_queue(dev);
}
```


然后就是发送函数。可以在这里将数据发送到红外接口。

```
static int pxa_irda_hard_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct pxa_irda *si = netdev_priv(dev);
    int speed = irda_get_next_speed(skb);
    ...
    //是否快速红外协议
    if (!IS_FIR(si)) {
        si->tx_buff.data = si->tx_buff.head;
        si->tx_buff.len  = async_wrap_skb(skb, si->tx_buff.data, si->tx_buff.
        truesize);
        //禁止中断，进入发送模式
        STIER = 0;
        STISR = IrSR_IR_TRANSMIT_ON | IrSR_XMODE_PULSE_1_6;
        //允许发送中断
        STIER = IER_UUE | IER_TIE;
    } else {
        unsigned long mtt = irda_get_mtt(skb);
        si->dma_tx_buff_len = skb->len;
        memcpy(si->dma_tx_buff, skb->data, skb->len);
        if (mtt)
            while ((unsigned)(OSCR_si->last_oscr)/4 < mtt)
                cpu_relax();
        //停止接收 DMA，禁止 FICP
        DCSR(si->rxdma) &= ~DCSR_RUN;
        ICCR0 = 0;
        //开始数据发送
        pxa_irda_fir_dma_tx_start(si);
        ICCR0 = ICCR0_ITR | ICCR0_TXE;
    }
    return 0;
}
```

在中断处理函数中分析红外端口的寄存器，判断是出现错误还是收到数据，并做相应地处理。STIIR 是 PXA27X 的红外中断源识别寄存器。

```
static irqreturn_t pxa_irda_sir_irq(int irq, void *dev_id)
{
    struct net_device *dev = dev_id;
```

```
    struct pxa_irda *si = netdev_priv(dev);
    int iir, lsr, data;
    iir = STIIR;
    switch  (iir & 0x0F) {
    case 0x06: //错误
    case 0x04: //数据到达
    case 0x0C: //超时
    case 0x02: //发送请求
    }
    return IRQ_HANDLED;//中断处理完毕
}
```

12.5 红外 SOCKET 通信

应用层的网络编程一般是利用套接字（Socket）实现的。在应用层，可以使用红外套接字（IrSock）进行通信。Linux 内核在 af_irda.c 文件中实现了 IrDA 套接字。如果仔细阅读这个源文件，就会理解红外通信是如何纳入套接字通信的范畴，并了解如何在网络 Socket 子系统中添加一种新的通信协议。内核中网络协议簇用下面的结构描述：

```
struct net_proto_family {
    int     family;
    int     (*create)(struct socket *sock, int protocol);
    short       authentication;
    short       encryption;
    short       encrypt net;
    struct module   *owner;
};
```

内核中所有的协议都注册在下面的数组中：

```
static struct net_proto_family *net_families[NPROTO];
```

套接字注册函数负责将新的协议加入到 net_families 中。系统中注册的协议是有一定数量的，这个数量就是 NPROTO。

```
int sock_register(struct net_proto_family *ops)
{
    int err;
    if (ops->family >= NPROTO) {
        printk(KERN CRIT "protocol %d >= NPROTO(%d)\n", ops->family, NPROTO);
        return -ENOBUFS;
    }
    net_family_write_lock();
    err = -EEXIST;
```

```
    if (net_families[ops->family] == NULL) {
        net_families[ops->family]=ops;
        err = 0;
    }
    net_family_write_unlock();
    printk(KERN_INFO "NET: Registered protocol family %d\n",ops->family);
    return err;
}
```

回头来看 af_irda.c，红外协议的结构如下：

```
static struct net_proto_family irda_family_ops = {
    .family = PF_IRDA,
    .create = irda_create,
    .owner = THIS_MODULE,
};
```

irda_create 函数其实是在创建一个 Socket 的时候调用的，它的主要作用是分配一个 struct sock，并设置这个 sock 的 Socket 函数集合。

```
static int irda_create(struct socket *sock, int protocol)
{
    struct sock *sk;
    struct irda_sock *self;
    IRDA_DEBUG(2, "%s()\n", __FUNCTION__);
    //检查协议是否得到支持
    switch (sock->type) {
    case SOCK_STREAM:
    case SOCK_SEQPACKET:
    case SOCK_DGRAM:
        break;
    default:
        return -ESOCKTNOSUPPORT;
    }
    if ((sk = sk_alloc(PF_IRDA, GFP_ATOMIC, 1, NULL)) == NULL)
        return -ENOMEM;
       self = sk->sk_protinfo = kmalloc(sizeof(struct irda_sock), GFP_ATOMIC);
    if (self == NULL) {
        sk_free(sk);
        return -ENOMEM;
    }
    memset(self, 0, sizeof(struct irda_sock));
    IRDA_DEBUG(2, "%s() : self is %p\n", __FUNCTION__, self);
    init_waitqueue_head(&self->query_wait);
    //初始化 sk 结构
    sock_init_data(sock, sk);
```

```
    sk_set_owner(sk, THIS_MODULE);
    sk->sk_family = PF_IRDA;
    sk->sk_protocol = protocol;
    self->sk = sk;
    //根据协议，设置操作函数集合
    switch (sock->type) {
    case SOCK_STREAM:
        sock->ops = &irda_stream_ops;
        self->max_sdu_size_rx = TTP_SAR_DISABLE;
        break;
    case SOCK_SEQPACKET:
        sock->ops = &irda_seqpacket_ops;
        self->max_sdu_size_rx = TTP_SAR_UNBOUND;
        break;
    case SOCK_DGRAM:
        switch (protocol) {
        case IRDAPROTO_UNITDATA:
            sock->ops = &irda_dgram_ops;
            self->max_sdu_size_rx = TTP_SAR_UNBOUND;
            break;
        default:
            ERROR("%s: protocol not supported!\n", __FUNCTION__);
            return -ESOCKTNOSUPPORT;
        }
        break;
    default:
        return -ESOCKTNOSUPPORT;
    }
    self->ckey = irlmp_register_client(0, NULL, NULL, NULL);
    self->mask.word = 0xffff;
    self->rx_flow = self->tx_flow = FLOW_START;
    self->nslots = DISCOVERY_DEFAULT_SLOTS;
    self->daddr = DEV_ADDR_ANY;
    self->saddr = 0x0;
    return 0;
}
```

当协议类型为 SOCK_STREAM 时，将使用 irda_stream_ops 操作函数集合。其他的协议也有自己的操作集合，分别是 irda_seqpacket_ops 和 irda_dgram_ops。

```
static struct proto_ops SOCKOPS_WRAPPED(irda_stream_ops) = {
    .family =    PF_IRDA,
    .owner =     THIS_MODULE,
    .release =   irda_release,
    .bind =      irda_bind,
    .connect =   irda_connect,
```

```
    .socketpair =   sock no socketpair,
    .accept =   irda_accept,
    .getname =  irda_getname,
    .poll =     irda_poll,
    .ioctl =    irda_ioctl,
    .listen =   irda_listen,
    .shutdown = irda_shutdown,
    .setsockopt =   irda setsockopt,
    .getsockopt =   irda getsockopt,
    .sendmsg =  irda sendmsg,
    .recvmsg =  irda recvmsg stream,
    .mmap =     sock no mmap,
    .sendpage = sock_no_sendpage,
};
```

在应用层使用红外网络套接字，与一般的网络通信最大的区别是地址结构：

```
struct sockaddr_irda {
    sa_family_t sir_family; //协议类型，一般是 AF_IRDA
    _u8     sir_lsap_sel;   //链接服务访问点选择因子，0~127
    _u32    sir_addr;       //设备地址
    char    sir_name[25];   //服务名，通常:IrDA:TinyTP
};
```

注意上面的 AF_IRDA 与 PF_IRDA 其实是一个值。大多数的网络通信都是基于客户机/服务器模式。服务器建立侦听，等待连接。客户机发起连接，发送服务请求。下面是红外通信服务器的典型代码：

```
int   serversock, clientsock;
struct  sockaddr_irda   address={0},cli_addr={0};
char   rcvBuffer[100];
int len=0;
address. sir family= AF IRDA;
address. sir lsap sel=0;
address. sir addr=DEV ADDR ANY;
strcpy(address. sir name,"IrDA:TinyTP");
if ((serversock = socket (AF_IRDA,SOCK_STREAM,0))==-1)//创建 Socket
{
    perron("socket");
    return -1;
}
if (bind(serversock,(struct sockaddr*)&address, sizeof (address)) ==-1)
//绑定地址
{
    perron("bind ");
    close(serversock);
```

```
      return  -1;
    }
    if  (listen(serversock,5)==-1)//开始监听
    {
      perron("listen ");
      close(serversock);
      return  -1;
    }
    if  ((clientsock=accept(serversock,(struct  sockaddr*)&cli_addr,  0))==-1)
    //等待接收客户连接
    {
      perron("accept ");
      close(serversock);
      return  -1;
    }
    if  (len=recv(clientsock,  rcvBuffer,  10,0)==-1)  //接收数据
    {
      perron("send ");
      close(serversock);
      return  -1;
    }
    closesocket   (clientsock);//关闭连接
    closesocket   (serversock);
    return   0;
```

下面是红外通信客户端的典型代码：

```
int    clientsock;
struct  sockaddr_irda   address={0};
char   sendBuffer[100];
int len=10;
address. sir family= AF IRDA;
address. sir lsap sel=0;
address. sir addr=DEV ADDR ANY;
strcpy(address. sir name,"IrDA:TinyTP");
if  ((clientsock = socket  (AF_IRDA,SOCK_STREAM,0))==-1)  //创建 Socket
{
     perron("socket");
     return  -1;
}
if  (connect(clientsock,(struct sockaddr*)&address,  sizeof  (address))  ==-1)
//连接服务器
{
     perron("connect ");
     close(clientsock);
     return  -1;
```

```
    }
    if (send(clientsock, sendBuffer, 10,0)==-1) //发送数据
    {
       perron("send ");
       close(clientsock);
       return  -1;
    }
    closesocket (clientsock);//关闭连接
    return   0;
```

第13章 音频设备驱动

音频设备本质上是一种字符型设备，但在 Linux 内核中却把它放在/drivers 目录之外。Linux 内核中包含两大音频体系：一是 OSS；一是 ALSA。开发 Linux 音频设备驱动要遵循这两个规范其中之一，否则驱动将不支持目前 Linux 下常用的多媒体播放器。

13.1 Linux 音频体系

在 Linux 2.6 中，音频驱动放到/sound 目录下，该目录下面的大部分的代码来自 ALSA 音频体系。Linux 原来的 OSS 音频体系由于其未完全开放代码，已经逐渐被冷落，ALSA 成为 Linux 2.6 内核中默认的标准音频驱动程序。

OSS 音频体系的 API 定义在<soundcard.h>中，它支持标准的文件操作，提供了 PCM、MIDI 接口。图 13.1 是 OSS 音频体系的基本框架：

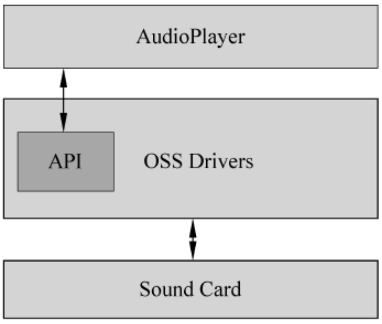


图 13.1 OSS 音频体系的基本框架

OSS 音频体系主要提供了如下的设备文件接口。

（1）/dev/mixer：混音器设备，主要是对声卡进行设置，比较设置 Speaker、MIC 和 MIDI 的音量、选择音源等。

（2）/dev/sndstat：测试用途，执行 cat /dev/sndstat 会显示声卡驱动的信息。

（3）/dev/dsp 和/dev/audio：音频数据通道，读这个设备就相当于录音，写这个设备就相当于放音。/dev/dsp 与/dev/audio 之间的区别在于采样的编码不同。/dev/audio 使用 μ 律编码，/dev/dsp 使用 8-bit（无符号）线性编码。

（4）/dev/sequencer：声音合成（synthesizer）设备，给电子（MIDI）音乐应用程序使用的。声音合成（synthesizer）设备的功能是把 MIDI 转换成波型数据。

（5）/dev/music：类似于/dev/sequencer。

（6）/dev/midi：MIDI 总线端口的底层接口，它的工作方式很像一个 TTY（character terminal），所有发送给它的数据立即传递到 MIDI 端口。

ALSA 音频体系不仅提供了内核驱动模块，还专门为应用程序的编写提供了方便的函数库，并提供了 MIGI 接口。它有如下特点：

（1）支持所有类型的音频接口，从普通的声卡到专业的音频设备。

（2）完全模块化的声卡驱动程序。

（3）SMP 和线程安全的设计。

（4）一个用户空间的函数库，提供了高层次的编程接口，从而简化了应用程序的开发。

（5）支持较老的 OSS API，兼容大多数 OSS 应用程序。

ALSA 音频体系包含的组件如表 13.1 所示：

表 13.1　ALSA 音频体系包含的组件

组件	说明
alsa-driver	内核驱动程序，包括硬件相关的和一些公共代码
alsa-lib	用户空间的函数库，这是给应用程序使用的。要包含头文件 asoundlib.h，链接共享库 libasound.so。
alsa-plugins	提供了多个插件
alsa-utils	一些基于 alsa 的命令行小程序，可以作为示例代码参考
alsa-tools	一些小工具，比如 vxloader 可以用来加载 Firmware
alsa-firmware	一些音频设备的固件程序。一些音频设备需要内核在合适的时候将固件程序下载到自己的 RAM 里
alsa-oss	与 OSS 兼容的代码
pyalsa	ALSA 音频 API 的 Python 语言的封装

ALSA 音频体系对应用层提供了如下 API 接口：

（1）设备信息接口（/proc/asound）。

（2）设备控制接口（/dev/snd/controlCX）。

（3）混音器设备接口（/dev/snd/mixerCXDX）。

（4）PCM 设备接口（/dev/snd/pcmCXDX）。

（5）原始 MIDI 设备接口（/dev/snd/midiCXDX）。

（6）声音合成（synthesizer）设备接口（/dev/snd/seq）。

（7）定时器接口（/dev/snd/timer）。

13.2　UDA1341TS 音频原理

UDA1341TS 是由 Philips 公司生产的一款音频芯片。UDA1341TS 提供标准的 IIS 接口，（又称 I^2S 接口），可以直接和 S3C2410X 的 IIS 引脚连接。另外，此芯片还提供标准的 L3 接口、麦克风和扬声器接口。L3 接口的引脚分别连到 S3C2410X 的三个 GPIO 输出引脚上，通过 GPIO 模拟 L3 接口。UDA1341TS 音频芯片集成数字化音频和混频器功能。数字化音频功能可以播放数字化声音或录制声音，因为包括这个功能，所以常把此类芯片称为 CODEC 设备。混频器用来控制各种输入/输出的音量大小，在 UDA1341TS 芯片中通过 L3 接口的进行控制。图 13.2 是 UDA1341TS 与 S3C2410X 连接的典型原理。

通过 L3 接口，S3C2410X 可以对 UDA1341TS 进行各种设置，包括系统频率、电源控制、音量、AGC 控制等。L3 接口包括L3DATA（数据线）、L3MODE（模式线）、L3CLOCK（时钟线）三根线。通过 L3 接口传送的信息分为两种模式：一种是地址模式，另一种是数据模式。

地址模式用于为后续数据传送选择一个设备，并定义一个目标寄存器。地址模式的特征是 L3MODE 为低，L3CLOCK 线上出现 8 个时钟脉冲，同时在数据线上伴随 8 个数据位。这 8 个数

据位中[7:2]位代表地址，[1:0]位代表数据类型。L3 接口的地址模式如图 13.3 所示，[1:0]位的含义如表 13.2 所示。

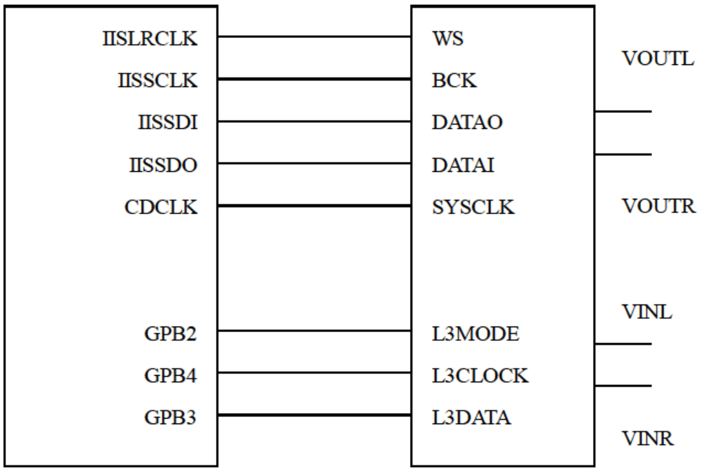


图 13.2　UDA1341TS 与 S3C2410X 连接的典型原理图

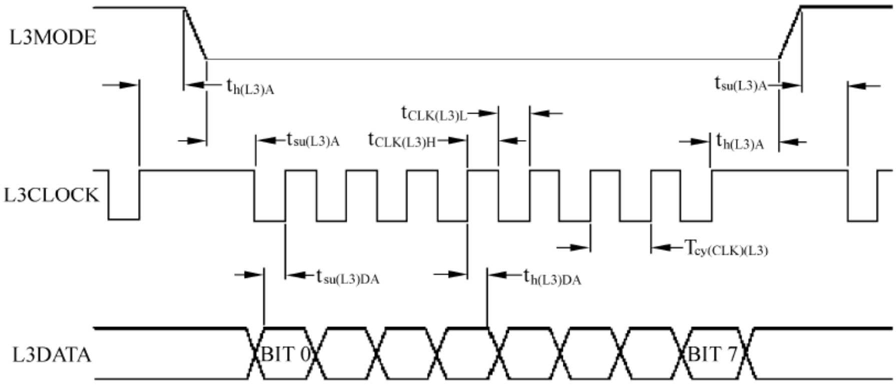


图 13.3　L3 接口的地址模式

表 13.2　1 到 0 位的含义

Bit[1:0]位	模式	意义
00	DATA0	直接寻址寄存器（音量、音部、峰值检测、静音等） 扩展寻址寄存器（数字混音、AGC 控制、输入增益等）
01	DATA1	峰值读出
10	STATUS	复位、系统时钟、数据输入格式、DC 过滤、输入增益开关、输出增益开关、电源控制
11	未用	

数据模式的特征是 L3MODE 为高。数据以 8 比特为一组进行收发。一个地址模式后可以传送多组数据。典型的数据模式时序如图 13.4 所示：

如果想对 UDA1341TS 的寄存器进行操作。需要搞清楚它的格式。比如要设置 DATA0 的直接

寻址寄存器，发送的数据的一部分用来选择寄存器地址，要设置音量，[7:6]位必须等于 00。[5:0]位是音量值。整个过程应该是先在地址模式中发送[1:0]位=00 的数据，然后在数据模式发送[7:6]位=0 的数据，可以参考表 13.3 所示的数据。

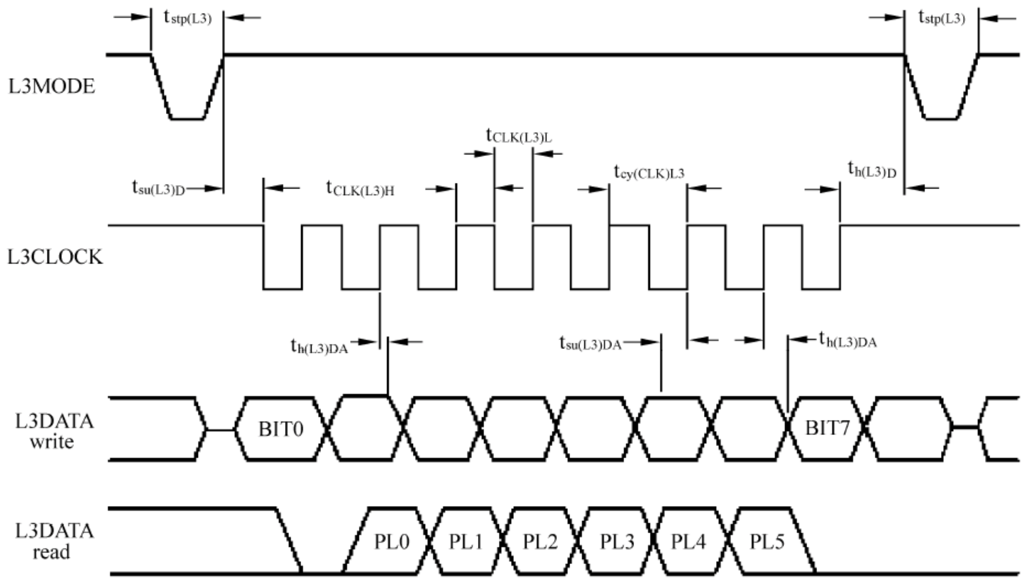


图 13.4　L3 接口的数据模式

表 13.3　DATA0 寄存器直接控制

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	REGISTER SELECTED
0	0	VC5	VC4	VC3	VC2	VC1	VC0	VC=volume control(6 bit)
0	1	BB3	BB2	BB1	BB0	TR1	TR0	BB=bass boost(4bits)
								TR=treble (2 bits)
1	0	PP	DE1	DE0	MT	M1	M0	PP=peak detection position
								DE=de-emphasis(2 bits)
								MT=mute
								M=mode swtich(2 bits)
1	1	0	0	0	EA2	EA1	EA0	EA=extended address(3 bits)
1	1	1	ED4	ED3	ED2	ED1	ED0	ED=extended data(5 bits)

13.3　S3C2410X 的音频接口

S3C2410X 的 IIS 控制器可以被用作 CODEC 接口，连接一个外部 8/16 比特的音频编解码芯片。IIS 接口提供了 DMA 模式，并且能够同时发送与接收数据。S3C2410X 通过 IIS 接口以 DMA 方式发送与接收 UDA1341TS 的音频数据，并通过 L3 通道对 UDA1341TS 进行控制。首先看看 IIS 控制器的主要寄存器：

（1）IIS 控制寄存器，如表 13.4 所示。

表 13.4　IIS 控制寄存器位定义

IISCON	位	描述	初始化
Left/Right channel index(Read only)	[8]	0 =左； 1 =右	1
Transmit FIFO ready flag(Read only)	[7]	0 =发送 FIFO 空； 1 =发送 FIFO 非空	0
Receive FIFO ready flag(Read only)	[6]	0 =接收 FIFO 满； 1 =接收 FIFO 未满	0
Transmit DMA service request	[5]	0 =禁止； 1 =允许	0
Receive DMA service request	[4]	0 =禁止； 1 =允许	0
Transmit channel idle command	[3]	空闲状态下 IISLRCK 不活动 0 =不空闲；1 =空闲	0
Receive channel idle command	[2]	空闲状态下 IISLRCK 不活动 0 =不空闲；1 =空闲	0
IIS prescaler	[1]	0 =禁止； 1 =允许	0
IIS interface	[0]	0 =停止 IIS； 1 =启动 IIS	0

（2）IIS 模式寄存器，如表 13.5 所示。

表 13.5　IIS 模式寄存器位定义

IISMOD	位	描述	初始化
Master/slave mode select	[8]	0 = 主模式（IISLRCK 和 IISCLK 为输出）； 1 = 从模式（IISLRCK 和 IISCLK 为输入）	0
Transmit/receive mode select	[7:6]	00 = 不传输；01 = 接收模式； 10 = 发送模式；11 = 接收和发送模式	00
Active level of left/right channel	[5]	0 =低选左声道，高选右声道； 1 =高选左声道，低选右声道	0
Serial interface format	[4]	0 = IIS 兼容格式； 1 = MSB（左）格式	0
Serial data bit per channel	[3]	0 = 8-bit； 1 = 16-bit	0
Master clock frequency select	[2]	0 = 256fs 1 = 384fs	0
Serial bit clock frequency select	[1:0]	00 = 16fs；01 = 32fs；10 = 48fs；11 = N/A	00

注意 fs 为采样频率。

（3）IIS 预分频寄存器，如表 13.6 所示。

表 13.6　IIS 预分频寄存器位定义

IISPSR	位	描述	初始化
Prescaler control A	[9:5]	N= 0～31 内部时钟为主时钟分频得到，分频因子为 N+1	00000
Prescaler control B	[4:0]	N= 0～31 外部时钟为主时钟分频得到，分频因子为 N+1	00000

（4）IIS FIFO 控制寄存器，如表 13.7 所示。

表 13.7　IIS FIFO 控制寄存器位定义

IISFCON	位	描述	初始化
Transmit FIFO access mode select	[15]	0 = 正常； 1 = DMA	0
Receive FIFO access mode select	[14]	0 =正常； 1 = DMA	0
Transmit FIFO	[13]	0 = 禁止；1 =允许	0
Receive FIFO	[12]	0 = 禁止；1 =允许	0
Transmit FIFO data count(Read only)	[11:6]	数据计数 =0 ～ 32	000000
Receive FIFO data count(Read only)	[5:0]	数据计数 =0 ～ 32	000000

S3C2410X 的 DMA 支持 4 类 DMA 传输：(1) 系统总线到系统总线（ASB/AHB to ASB/AHB）；(2）系统总线到外设总线（ASB/AHB to APB）；(3）外设总线到系统总线（APB to ASB/AHB）；(4) 外设总线到外设总线（APB to APB）。S3C2410X 共有 4 条 DMA 通道，每条通道 5 个请求源，如表 13.8 所示。

表 13.8　4 条 DMA 通道与其对应的请求源

通道	源 1	源 2	源 3	源 4	源 5
Ch0	nXDREQ0	UART0	SDI	Timer	USB device EP1
Ch1	nXDREQ1	UART1	I2SSDI	SPI0	USB device EP2
Ch2	I2SSDO	I2SSDI	SDI	Timer	USB device EP3
Ch3	UART2	SDI	SPI1	Timer	USB device EP4

S3C2410X 的 DMA 控制器的重要控制寄存器如下：

（1）DMA 源地址寄存器，如表 13.9 所示。

表 13.9　DMA 源地址寄存器位定义

DISRCn	位	描述	初始化
S_ADDR	[30:0]	传输数据的源地址	0x00000000

（2）DMA 源控制寄存器，如表 13.10 所示。

表 13.10　DMA 源控制寄存器位定义

DISRCCn	位	描述	初始化
LOC	[1]	用来选择源： 0=系统总线 AHB； 1=外部总线 APB	0
INC	[0]	用来选择地址增长方式： 0=每次数据传输后地址增加数据传输的尺寸；1=地址保持不变	0

（3）DMA 目标地址寄存器，如表 13.11 所示。

表 13.11　DMA 目标地址寄存器位定义

DIDSTn	位	描述	初始化
S_ADDR	[30:0]	传输数据的目标地址	0x00000000

（4）DMA 目标控制寄存器，如表 13.12 所示。

表 13.12　DMA 目标控制寄存器位定义

DIDSTCn	位	描述	初始化
LOC	[1]	用来选择源： 0=系统总线 AHB； 1=外部总线 APB	0
INC	[0]	用来选择地址增长方式： 0=每次数据传输后地址增加数据传输的尺寸；1=地址保持不变	0

（5）DMA 控制寄存器，如表 13.13 所示。

表 13.13　DMA 控制寄存器位定义

DCONn	位	描述	初始化
DMD_HS	[31]	0=命令模式；1=握手模式	0
SYNC	[30]	0=DREQ 和 DACK 与 PCLK 同步； 1=DREQ 和 DACK 与 HCLK 同步	0
INT	[29]	0=禁止 CURR_TC 中断； 1=传输完成产生中断（CURR_TC=0）	0
TSZ	[28]	选择原子传输的尺寸	0
SERVMODE	[27]	选择服务模式	0
HWSRCSEL	[26:24]	选择每个 DMA 通道的请求源： DCON0 中 000=nXDREQ0；001=UART0；010=SDI；011=Timer；100=USB EP1 DCON1 中 000=nXDREQ1；001=UART1；010=I2SSDI；011=SPI；100=USB EP2 DCON2 中 000=I2SSDO；001=I2SSDI；010=SDI；011=Timer；100=USB EP3 DCON3 中 000=UART2；001=SDI；010=SPI；011=Timer；100=USB EP4	00

续表

DCONn	位	描述	初始化
SWHW_SEL	[23]	0=软请求模式；1=DMA 源由 Bit[26:24]选择	0
RELOAD	[22]	自动重载选择	0
DSZ	[21]	数据传输的单元 00 = Byte；　01 = Half word； 10 = Word；11 = 保留	00
TC	[19:0]	初始传输计数 注意，实际传输字节为 DSZ×TSZ×TC。 这个值只有当 CURR_SRC=0 和 DMA ACK =1 时才装入 CURR_SRC	00000

下面来看看与 DMA 通信相关的主要函数（Linux/arch/arm/mach-s3c2410/dma.c）。内核在初始化的时候调用下面的函数，初始化 DMA 通道：

```
static int __init s3c2410_init_dma(void)
{
    struct s3c2410_dma_chan *cp;
    int channel;
    int ret;
    printk("S3C24XX DMA Driver, (c) 2003-2004,2006 Simtec Electronics\n");
    //物理地址的映射
    dma_base = ioremap(S3C24XX_PA_DMA, 0x200);
    if (dma_base == NULL) {
        printk(KERN_ERR "dma failed to remap register block\n");
        return -ENOMEM;
    }
    //注册系统设备
    ret = sysdev_class_register(&dma_sysclass);
    if (ret != 0) {
        printk(KERN_ERR "dma sysclass registration failed\n");
        goto err;
    }
    dma_kmem = kmem_cache_create("dma_desc", sizeof(struct s3c2410_dma_buf), 0,
                SLAB_HWCACHE_ALIGN,s3c2410_dma_cache_ctor, NULL);
    if (dma_kmem == NULL) {
        printk(KERN_ERR "dma failed to make kmem cache\n");
        ret = -ENOMEM;
        goto err;
    }
    //逐个初始化 DMA 通道
    for (channel = 0; channel < S3C2410_DMA_CHANNELS; channel++) {
        cp = &s3c2410_chans[channel];
        memset(cp, 0, sizeof(struct s3c2410_dma_chan));
        cp->number = channel;           //DMA 通道号
        cp->irq    = channel + IRQ_DMA0;//中断号
```

```
        cp->regs   = dma_base + (channel*0x40);
        cp->stats  = &cp->stats_store;      //统计信息
        cp->stats_store.timeout_shortest = LONG_MAX;
        cp->load_timeout = 1<<18;           //超时
        cp->dev.cls = &dma_sysclass;
        cp->dev.id  = channel;
        ret = sysdev_register(&cp->dev);
        printk("DMA channel %d at %p, irq %d\n",cp->number, cp->regs, cp->irq);
    }
    return 0;
 err:
    kmem_cache_destroy(dma_kmem);
    iounmap(dma_base);
    dma_base = NULL;
    return ret;
}
```

s3c2410_dma_devconfig 函数的作用是设置 S3C2410X 的 DMA 寄存器。

```
int s3c2410_dma_devconfig(int channel,enum s3c2410_dmasrc source,int hwcfg,
unsigned long devaddr)
{
    struct s3c2410_dma_chan *chan = lookup_dma_channel(channel);
    if (chan == NULL)
        return -EINVAL;
    pr_debug("%s: source=%d, hwcfg=%08x, devaddr=%08lx\n",
         __FUNCTION__, (int)source, hwcfg, devaddr);
    chan->source = source;
    chan->dev_addr = devaddr;
    switch (source) {
    case S3C2410_DMASRC_HW:
        //请求源是硬件
        pr_debug("%s: hw source, devaddr=%08lx, hwcfg=%d\n",
             __FUNCTION__, devaddr, hwcfg);
        dma_wrreg(chan, S3C2410_DMA_DISRCC, hwcfg & 3);//设置 DMA 源控制寄存器
        dma_wrreg(chan, S3C2410_DMA_DISRC,  devaddr);  //设置 DMA 源地址寄存器
        dma_wrreg(chan, S3C2410_DMA_DIDSTC, (0<<1) | (0<<0)); //设置 DMA 目标控制
                                                               //寄存器
        chan->addr_reg = dma_regaddr(chan, S3C2410_DMA_DIDST);
        return 0;
    case S3C2410_DMASRC_MEM:
        //请求源为内存
        pr_debug( "%s: mem source, devaddr=%08lx, hwcfg=%d\n",
              __FUNCTION__, devaddr, hwcfg);
        dma_wrreg(chan, S3C2410_DMA_DISRCC, (0<<1) | (0<<0));
        dma_wrreg(chan, S3C2410_DMA_DIDST,  devaddr);
        dma_wrreg(chan, S3C2410_DMA_DIDSTC, hwcfg & 3);
```

```
        chan->addr reg = dma regaddr(chan, S3C2410 DMA DISRC);
        return 0;
    }
    printk(KERN_ERR "dma%d: invalid source type (%d)\n", channel, source);
    return -EINVAL;
}
```

s3c2410_dma_request 用来对 channel 指定的 DMA 通道进行设置，并申请 DMA 中断。

```
int s3c2410_dma_request(unsigned int channel,struct s3c2410_dma_client
*client,void *dev)
{
    struct s3c2410_dma_chan *chan;
    unsigned long flags;
    int err;
    pr debug("dma%d: s3c2410 request dma: client=%s, dev=%p\n",
         channel, client->name, dev);
    local_irq_save(flags);
    chan = s3c2410_dma_map_channel(channel);//返回第 channel 个 DMA 通道
    if (chan == NULL) {
        local_irq_restore(flags);
        return -EBUSY;
    }
    dbg_showchan(chan);//设置 DMA 的寄存器
    chan->client = client;
    chan->in_use = 1;
    if (!chan->irq_claimed) {
    pr debug("dma%d: %s : requesting irq %d\n",channel,  FUNCTION , chan->irq);
        chan->irq claimed = 1;
        local irq restore(flags);
        //申请 DMA 中断处理
        err = request_irq(chan->irq, s3c2410_dma_irq, IRQF_DISABLED,
                  client->name, (void *)chan);
        local_irq_save(flags);
        if (err) {
            chan->in use = 0;
            chan->irq claimed = 0;
            local_irq_restore(flags);
            printk(KERN_ERR "%s: cannot get IRQ %d for DMA %d\n",
                  client->name, chan->irq, chan->number);
            return err;
        }
        chan->irq enabled = 1;
    }
    local irq restore(flags);
    return 0;
}
```

DMA 的启动是通过 s3c2410_dma_enqueue，将数据放入待发送队列。

```
int s3c2410_dma_enqueue(unsigned int channel, void *id,dma_addr_t data, int size)
{
    struct s3c2410_dma_chan *chan = lookup_dma_channel(channel);
    struct s3c2410_dma_buf *buf;
    unsigned long flags;
    if (chan == NULL)return -EINVAL;
    pr debug("%s: id=%p, data=%08x, size=%d\n",
          FUNCTION , id, (unsigned int)data, size);
    buf = kmem_cache_alloc(dma_kmem, GFP_ATOMIC);
    if (buf == NULL) {
        pr_debug("%s: out of memory (%ld alloc)\n",_FUNCTION_,
        (long)sizeof(*buf));
        return -ENOMEM;
    }
    buf->next  = NULL;
    buf->data  = buf->ptr = data;   //设置数据地址
    buf->size  = size;              //设置尺寸
    buf->id    = id;                //DMA 客户 ID
    buf->magic = BUF MAGIC;
    local_irq_save(flags);
    if (chan->curr == NULL) {
        chan->curr = buf;
        chan->end  = buf;
        chan->next = NULL;
    } else {
        chan->end->next = buf;
        chan->end = buf;
    }
    if (chan->next == NULL)chan->next = buf;
    //检查 DMA 是否空闲
    if (chan->state == S3C2410_DMA_RUNNING) {
        if (chan->load_state == S3C2410_DMALOAD_1LOADED && 1) {
            if (s3c2410_dma_waitforload(chan, _LINE_) == 0) {
                printk(KERN_ERR "dma%d: loadbuffer:""timeout loading buffer\n",
                      chan->number);
                dbg_showchan(chan);
                local irq restore(flags);
                return -EINVAL;
            }
        }
        while (s3c2410 dma canload(chan) && chan->next != NULL) {
            s3c2410 dma loadbuffer(chan, chan->next);
        }
```

```
    } else if (chan->state == S3C2410_DMA_IDLE) {
        if (chan->flags & S3C2410_DMAF_AUTOSTART) {
          //启动 DMA 数据传输
            s3c2410_dma_ctrl(chan->number, S3C2410_DMAOP_START);
        }
    }
    local_irq_restore(flags);
    return 0;
}
```

13.4　UDA1341TS 驱动开发

UDA1341TS 和 S3C2410X 之间有两个通道，一个是 IIS 接口，另一个是 L3 接口。IIS 接口的数据传输可以通过 DMA 方式进行，L3 接口的时序就需要自己实现了。uda1341_l3_address 实现了通过 L3 接口发送地址。

```
static void uda1341_l3_address(u8 data)
{
    int i;
    unsigned long flags;
    local_irq_save(flags);
    s3c2410_gpio_setpin(S3C2410_GPB2,0);
    s3c2410_gpio_setpin(S3C2410_GPB4,1);
    udelay(1);
    for (i = 0; i < 8; i++) {
        if (data & 0x1) {
            s3c2410_gpio_setpin(S3C2410_GPB4,0);
            s3c2410_gpio_setpin(S3C2410_GPB3,1);
            udelay(1);
            s3c2410_gpio_setpin(S3C2410_GPB4,1);
        } else {
            s3c2410_gpio_setpin(S3C2410_GPB4,0);
            s3c2410_gpio_setpin(S3C2410_GPB3,0);
            udelay(1);
            s3c2410_gpio_setpin(S3C2410_GPB4,1);
        }
        data >>= 1;
    }
    s3c2410_gpio_setpin(S3C2410_GPB2,1);
    s3c2410_gpio_setpin(S3C2410_GPB4,1);
    local_irq_restore(flags);
}
```

uda1341_l3_data 实现了通过 L3 接口发送 8 比特数据。

```
static void uda1341 l3 data(u8 data)
{
    int i;
    unsigned long flags;
    local_irq_save(flags);
    udelay(1);
    for (i = 0; i < 8; i++) {
        if (data & 0x1) {
            s3c2410 gpio setpin(S3C2410 GPB4,0);
            s3c2410 gpio setpin(S3C2410 GPB3,1);
            udelay(1);
            s3c2410 gpio setpin(S3C2410 GPB4,1);
        } else {
            s3c2410_gpio_setpin(S3C2410_GPB4,0);
            s3c2410_gpio_setpin(S3C2410_GPB3,0);
            udelay(1);
            s3c2410_gpio_setpin(S3C2410_GPB4,1);
        }
        data >>= 1;
    }
    local irq restore(flags);
}
```

有了上面两个函数，就可以轻松控制 UDA1341TS 了。现在以设置音量为例，前面已经分析过发送的过程，先在地址模式中发送[1:0]位=00 的数据，然后在数据模式发送[7:6]位=0 的数据。

```
uda1341_l3_address(UDA1341_REG_DATA0);      //地址模式
uda1341_l3_data(DATA0 |DATA0_VOLUME(0x0)); //数据模式，发送音量
```

内核中提供了 register_sound_dsp 和 register_sound_mixer 用来注册音频设备。现在来分析开发 UDA1341TS 驱动的整体过程。

```
static int_init audio_init_dma(audio_stream_t * s, char *desc)
{
    int ret ;
    s3c2410 dmasrc t source;
    int hwcfg;
    unsigned long devaddr;
    dmach_t channel;
    int dcon;
    unsigned int flags = 0;
    if(s->dma_ch == DMA_CH2){//内存 DMA 源
        channel = 2;
        source = S3C2410 DMASRC MEM;
        hwcfg = 3;
        devaddr = 0x55000010;
```

```
            dcon = 0xa0800000;
            flags = S3C2410_DMAF_AUTOSTART;//自启动
            //设置 DMA 寄存器，并申请 DMA 通道
            s3c2410_dma_devconfig(channel, source, hwcfg, devaddr);
            s3c2410_dma_config(channel, 2, dcon);
            s3c2410_dma_set_buffdone_fn(channel, audio_dmaout_done_callback);
            s3c2410_dma_setflags(channel, flags);
            ret = s3c2410_dma_request(s->dma_ch, &s3c2410iis_dma_out, NULL);
            s->dma_ok = 1;
            return ret;
        }
        else if(s->dma_ch == DMA_CH1){//硬件 DMA 源
            channel =1;
            source =S3C2410_DMASRC_HW;
            hwcfg =3;
            devaddr = 0x55000010;
            dcon = 0xa2900000;
            flags = S3C2410_DMAF_AUTOSTART; ;//自启动
            //设置 DMA 寄存器，并申请 DMA 通道
            s3c2410_dma_devconfig(channel, source, hwcfg, devaddr);
            s3c2410_dma_config(channel, 2, dcon);
            s3c2410_dma_set_buffdone_fn(channel, audio_dmain_done_callback);
            s3c2410_dma_setflags(channel, flags);
            ret = s3c2410_dma_request(s->dma_ch, &s3c2410iis_dma_in, NULL);
            s->dma_ok =1;
            return ret ;
        }
        else
            return 1;
    }
```

需要在驱动中提供一个 DSP 设备和一个 MIXER 设备，以使设备支持 OSS 和 ALSA 体系。这两个设备有各自的操作函数集合。输出通道从内存到硬件，所以采用内存 DMA 源；而输入通道是从 IIS 到内存，所以采用 IIS 硬件源。

```
    if (audio_init_dma(&output_stream, "UDA1341 out")) {
        audio_clear_dma(&output_stream,&s3c2410iis_dma_out);
        printk( KERN_WARNING AUDIO_NAME_VERBOSE
                ": unable to get DMA channels\n" );
        return -EBUSY;
    }
    printk( "success to get DMA channels 1\n " );
    input_stream.dma_ch = DMA_CH1;
    if (audio_init_dma(&input_stream, "UDA1341 in")) {
        audio_clear_dma(&input_stream,&s3c2410iis_dma_in);
```

```
        printk( KERN WARNING AUDIO NAME VERBOSE
                "; unable to get DMA channels\n" );
        return -EBUSY;
    }
    audio_dev_dsp = register_sound_dsp(&smdk2410_audio_fops, -1);
    audio_dev_mixer = register_sound_mixer(&smdk2410_mixer_fops, -1);
```

这里又出现了熟悉的 struct file_operations：

```
static struct file operations smdk2410 audio fops = {
    llseek: smdk2410 audio llseek,
    write: smdk2410 audio write,
    read: smdk2410_audio_read,
    poll: smdk2410_audio_poll,
    ioctl: smdk2410_audio_ioctl,
    open: smdk2410_audio_open,
    release: smdk2410_audio_release
};
static struct file operations smdk2410 mixer fops = {
    ioctl: smdk2410 mixer ioctl,
    open: smdk2410 mixer open,
    release: smdk2410 mixer release
};
```

打开函数主要是权限检查、引用计数管理、设置默认参数。

```
static int smdk2410_audio_open(struct inode *inode, struct file *file)
{
    int cold = !audio active;
    //对设备的读写权限进行处理
    if ((file->f flags & O ACCMODE) == O RDONLY) {
        if (audio rd refcount || audio wr refcount)
            return -EBUSY;
        audio_rd_refcount++;
    } else if ((file->f_flags & O_ACCMODE) == O_WRONLY) {
        if (audio_wr_refcount)
            return -EBUSY;
        audio_wr_refcount++;
    } else if ((file->f_flags & O_ACCMODE) == O_RDWR) {
        if (audio rd refcount || audio wr refcount)
            return -EBUSY;
        audio rd refcount++;
        audio wr refcount++;
    } else
        return -EINVAL;
    //设置默认参数
        if (cold) {
```

```
                audio_rate = AUDIO_RATE_DEFAULT;
                audio_channels = AUDIO_CHANNELS_DEFAULT;
                audio_fragsize = AUDIO_FRAGSIZE_DEFAULT;
                audio_nbfrags = AUDIO_NBFRAGS_DEFAULT;
                if ((file->f_mode & FMODE_WRITE)){
                    init_s3c2410_iis_bus_tx();
                    audio_clear_buf(&output_stream);
                }
                if ((file->f_mode & FMODE_READ)){
                    init_s3c2410_iis_bus_rx();
                    audio_clear_buf(&input_stream);
                }
            }
            return 0;
    }
```

音频播放就是调用 smdk2410_audio_write。这个函数将用户数据复制到内核地址空间，然后将这些数据放入 DMA 发送队列中。音频录音的过程与放音的过程相似，只是数据传输的方向相反。

```
    static ssize_t smdk2410_audio_write(struct file *file, const char *buffer,size_t
    count, loff_t * ppos)
    {
        const char *buffer0 = buffer;
        audio_stream_t *s = &output_stream;     //指向输出通道
        int chunksize, ret = 0;
        DPRINTK("audio_write : start count=%d\n", count);
        switch (file->f_flags & O_ACCMODE) {    //判断读写权限
            case O_WRONLY:
            case O_RDWR:
                break;
            default:
                return -EPERM;
        }
        if (!s->buffers && audio_setup_buf(s)) //建立缓冲
            return -ENOMEM;
        count &= ~0x03;
        while (count > 0) {
            audio_buf_t *b = s->buf;
            if (file->f_flags & O_NONBLOCK) {   //非阻塞
                ret = -EAGAIN;
                if (down_trylock(&b->sem))      //立即返回的锁获取
                    break;
            } else {
                ret = -ERESTARTSYS;
```

```
            if (down interruptible(&b->sem))
                break;
        }
        //根据声道的数量来复制数据
        if (audio_channels == 2) {
            chunksize = s->fragsize - b->size;
            if (chunksize > count)
                chunksize = count;
            DPRINTK("write %d to %d\n", chunksize, s->buf idx);
            if (copy from user(b->start + b->size, buffer, chunksize)) {
                up(&b->sem);
                return -EFAULT;
            }
            b->size += chunksize;
        } else {
            chunksize = (s->fragsize - b->size) >> 1;
            if (chunksize > count)
                chunksize = count;
            DPRINTK("write %d to %d\n", chunksize*2, s->buf_idx);
            if (copy from user mono stereo(b->start + b->size,
               buffer, chunksize)) {
                   up(&b->sem);
                   return -EFAULT;
               }
               b->size += chunksize*2;
        }
        buffer += chunksize;
        count -= chunksize;
        if (b->size < s->fragsize) {
            up(&b->sem);
            break;
        }
     //将这些数据放入 DMA 发送队列
    if((ret = s3c2410_dma_enqueue(s->dma_ch, (void *) b, b->dma_addr, b->size))) {
            printk(PFX"dma enqueue failed.\n");
            return ret;
        }
        b->size = 0;
        NEXT_BUF(s, buf);
    }
    if ((buffer - buffer0))// //计算写入的实际数据量
        ret = buffer - buffer0;
    DPRINTK("audio write : end count=%d\n\n", ret);
    return ret;
}
```

13.5　音频应用层编程

要学会在 Linux 下进行音频编程，首先要理解三个概念：（1）采样频率，即将模拟声音波形进行数字化时，每秒钟抽取声波幅度样本的次数；（2）量化位数，即用多少比特来衡量音频信号的幅度；（3）声道属性，如单声道，双声道，立体声等。

13.5.1　OSS 音频编程接口

OSS 的接口定义在<soundcard.h>中。利用 OSS 进行音频开发的主要步骤包括：（1）打开设备文件，获取文件描述符；（2）使用 ioctl 对音频的参数进行设置；（3）播放或录音；（4）关闭设备文件。下面的例子告诉你如何设置音量。

```
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/soundcard.h>
int main() {
    int mixer_fd;
    if ((mixer_fd = open("/dev/mixer",O_RDONLY,0)) == -1) {
        exit(1);
    }
     int vol = 0x3F3F;
    ioctl(mixer_fd, MIXER_WRITE(SOUND_MIXER_VOLUME),&vol);
    close(mixer_fd);
    return 0;
}
```

OSS 中包含如表 13.14 所示的 MIXER 设备宏，用来设置音频的参数，OSS 中包含的 DSP 设备宏参数如表 13.15 所示。

表 13.14　OSS 中的 MIXER 设备宏

宏	描述
SOUND_MIXER_VOLUME	设置输出音量的大小
SOUND_MIXER_RECLEV	设置录音音量
SOUND_MIXER_TREBLE	设置所有声道输出高音的大小。
SOUND_MIXER_BASS	设置所有声道输出低音的大小。
SOUND_MIXER_PCM	设置音频设备（/dev/dsp）输出音量的大小。
SOUND_MIXER_MIC	设置从麦克风输入信号的大小。
SOUND_MIXER_CD	设置从 CD 输入信号的大小。
SOUND_MIXER_LINE	设置音频线输入信号的音量大小。

表 13.15　OSS 中的 DSP 设备宏

宏	描述
SNDCTL_DSP_RESET	停止设备，设备进入参数设定状态
SNDCTL_DSP_SPEED	采样频率设置，参数为 int speed
SNDCTL_DSP_SAMPLESIZE	采样尺寸设置
SNDCTL_DSP_CHANNELS	声道参数设置
SNDCTL_DSP_GETBLKSIZE	获取缓存块的大小
SNDCTL_DSP_SETFRAGMENT	设置缓冲区的内部缓存块的大小
SNDCTL_DSP_SYNC	处理对音频设备的同步访问
SNDCTL_DSP_SETFMT	音频格式设置

下面来看一个简单的音频录制与播放程序。

```
int audio fd;
//打开音频设备
if ((audio fd = open("/dev/dsp",omode,0)) == -1) {
    perror("/dev/dsp");
    exit(1);
}
//设置音频格式、声道数、采样频率
int format;
format = AFMT_S16_NE;
if (ioctl(audio_fd,SNDCTL_DSP_SETFMT, &format) == -1) {
    perror("SNDCTL_DSP_SETFMT");
    exit(1);
}
int channels = 2;
if (ioctl(audio fd, SNDCTL DSP CHANNELS, & channels) == -1) {
    perror("SNDCTL DSP CHANNELS");
    exit(1);
}
int speed = 44100;  //44.1 kHz
if (ioctl(audio_fd, SNDCTL_DSP_SPEED, &speed) == -1) {
    perror("SNDCTL_DSP_SPEED");
    exit(1);
}
int music fd;
signed short applicbuf[2048];
int count;
if (mode == PLAY) {//播放模式
    if ((music fd = open(filename, O RDONLY, 0)) == -1) {
        perror(filename);
        exit(1);
    }
```

```
        //从文件中读音频数据，直接发送到音频设备
         while ((count = read(music_fd, applicbuf, 2048)) > 0) {
             write(audio_fd, applicbuf, count);
         }
    } else {//录音模式
         if ((music_fd = open(filename, O_WRONLY | O_CREAT, 0)) == -1) {
            perror(filename);
            exit(1);
        }
         int mixer_fd;
        if ((mixer_fd = open("/dev/mixer", O_WRONLY)) == -1) {
            perror("open /dev/mixer error");
            exit(1);
        }
        //设置录音源
        int testchan = SOUND_MIXER_CD;
         int recsrc = (1 << testchan);
        if (ioctl(mixer_fd, SOUND_MIXER_WRITE_RECSRC,&recsrc) == -1) {
            perror("CD");
            exit(1);
         }
         int totalbyte= speed * channels * 2 * 60 * 3;
         int totalword = totalbyte/2;
         int total = 0;
         while (total != totalword) {
                 if (totalword - total >= 2048)
                     count = 2048;
                 else
                     count = totalword - total;
                 read(audio_fd, applicbuf, count);//录音
                 write(music_fd, applicbuf, count);//写入文件
                 total += count;
             }
             close(mixer_fd);
         }
         close(audio_fd);
         close(music_fd);
         return 0;
    }
```

13.5.2 ALSA 音频编程接口

alsalib 中提供了丰富的音频函数接口，支持音频设备、混音器设备、PCM 设备、原始 MIDI 设备、定时器等。alsalib 库使用起来其实和上面的 OSS 编程步骤差不多，仍然是打开设备后设置

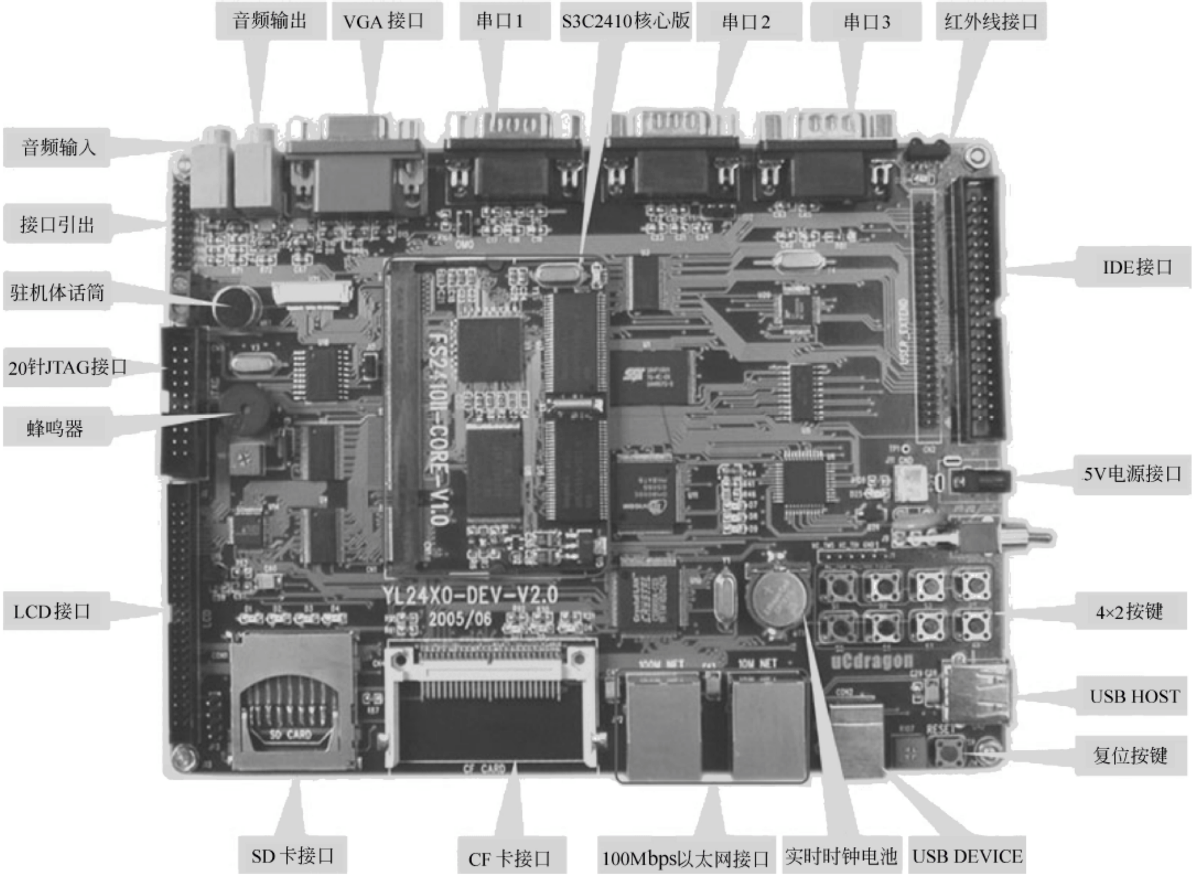
参数，最后读写数据。所有参数先存放到 snd_pcm_hw_params_t 结构中，设置好 snd_pcm_hw_params_t 结构后，将参数整体写入到设备驱动。参数设置完毕，就可以调用 snd_pcm_writei 和 snd_pcm_readi 来进行音频数据读写了。下面的程序演示了一个基本的 ALSA 播放器。

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>//ALSA 库
main (int argc, char *argv[])
{
    int i;
    int err;
    short buf[128];
    snd_pcm_t *playback_handle;
    snd_pcm_hw_params_t *hw_params;//硬件参数结构
    if((err=snd_pcm_open(&playback_handle,argv[1],
                                  SND_PCM_STREAM_PLAYBACK, 0)) < 0) {
            fprintf (stderr, "cannot open audio device %s (%s)\n", argv[1],
                 snd_strerror (err));
            exit (1);
    }
    if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
        fprintf (stderr, "cannot allocate hardware parameter structure (%s)\n"
,snd_strerror (err));
        exit (1);
    }
    if((err = snd_pcm_hw_params_any (playback_handle, hw_params)) < 0) {
        fprintf (stderr, "cannot initialize hardware parameter structure (%s)\n"
,snd_strerror (err));
        exit (1);
    }
    if((err = snd_pcm_hw_params_set_access (playback_handle, hw_params, SND_PCM_
ACCESS_RW_INTERLEAVED)) < 0) {
        fprintf (stderr, "cannot set access type (%s)\n",
                 snd_strerror (err));
        exit (1);
    }
   //设置音频格式
    if ((err = snd_pcm_hw_params_set_format (playback_handle, hw_params, SND_PCM_
FORMAT_S16_LE)) < 0) {
        fprintf (stderr, "cannot set sample format (%s)\n",
                 snd_strerror (err));
        exit (1);
    }
  //设置采样频率
    if ((err = snd_pcm_hw_params_set_rate_near (playback_handle,
```

```
                                                  hw params, 44100, 0)) < 0) {
            fprintf (stderr, "cannot set sample rate (%s)\n",
                     snd_strerror (err));
            exit (1);
        }
     //设置声道
        if ((err = snd_pcm_hw_params_set_channels (playback_handle, hw_params, 2))
    < 0) {
            fprintf (stderr, "cannot set channel count (%s)\n",
                     snd strerror (err));
            exit (1);
        }
     //将参数一起写入设备
        if ((err = snd_pcm_hw_params (playback_handle, hw_params)) < 0) {
            fprintf (stderr, "cannot set parameters (%s)\n",
                     snd_strerror (err));
            exit (1);
        }
        snd pcm hw params free (hw params);
        if ((err = snd pcm prepare (playback handle)) < 0) {
            fprintf (stderr, "cannot prepare audio interface for use (%s)\n",
                     snd strerror (err));
            exit (1);
        }
      //播放音频
        for (i = 0; i < 10; ++i) {
                if ((err = snd_pcm_writei (playback_handle, buf, 128)) != 128) {
                    fprintf (stderr, "write to audio interface failed (%s)\n",
                         snd_strerror (err));
                    exit (1);
                }
        }
      //关闭句柄
        snd pcm close (playback handle);
        exit (0);
    }
```

附录：深圳优龙科技 YL2410 开发板简介

YL2410 开发板采用核心板＋底板的模式，核心板为 6 层，底板为 4 层。核心板接口采用 DIMM－200 标准连接器，并且兼容 YL2440 核心板。核心板和底板的布局和走线经过专业人士精心设计，工作非常可靠，可稳定运行在 203MHz。外设非常丰富，功能强大，适用于各种手持设备、消费电子和工业控制设备的开发，YL2410 开发板见附录 1 图所示。



附录 1 图

YL2410 开发板硬件资源

中央处理器

- ❑ CPU：三星 S3C2410A，主频 203MHz。

外部存储器

- ❑ 内存：64MB。
- ❑ NOR Flash：2MB（SST39VF1601）。

- NAND Flash: 64MB(K9F1208,用户可自己更换为 16MB、32MB 或 128MB 的 Nand Flash)。

串口

- 两个五线异步串行口，波特率高达 115 200bps。
- 一个九线异步串行口，采用 ST16C550 扩展出来的，波特率高达 1.5Mbps。

网络接口

- 一个 10Mbps 网口，采用 CS8900Q3，带联接和传输指示灯。
- 一个 100Mbps 网口，采用 DM9000，带联接和传输指示灯。

USB 接口

- 一个 USB 1.1 HOST 接口，支持 WLAN，提供二进制代码（不免费提供源码）。
- 一个 USB 1.1 Device 接口。

红外通信口

- 一个 IRDA 红外线数据通信口。

CAN 总线接口

- 一个 CAN 总线接口，全面支持 CAN 2.0A 和 CAN 2.0B 协议。

音频接口

- 采用 IIS 接口芯片 UDA1341，一路立体声音频输出接口可接耳机或音箱。
- 支持录音，板子自带驻机体话筒可直接录音，另有一路话筒输入接口可接麦克风。

存储接口

- 一个 SD 卡接口。
- 一个 CF 卡接口（3.3V，接口信号均加了 74LVTH162245 驱动），工作在 TrueIDE 模式。
- 一个 IDE 接口（接口信号均加了 74LVTH162245 驱动），可直接挂接硬盘。

LCD 和触摸屏接口

- 板上集成了四线电阻式触摸屏接口的相关电路。
- 一个 50 芯 LCD 接口引出了 LCD 控制器的全部信号，并且这些信号引脚都加了 74LVTH162245 驱动，所以 LCD 输出更加稳定可靠。
- 标准配置为 256k 色 240×320/3.5 英寸 TFT 液晶屏，带触摸屏。
- 支持黑白、4 级灰度、16 级灰度、256 色、4096 色 STN 液晶屏，尺寸从 3.5～12.1 寸，屏幕分辨率可达到 1024×768 像素。
- 板上引出一个 5V 电源输出接口，可为大尺寸 TFT 液晶屏的 5V CCFL 背光模块供电。

VGA 接口

- 一个标准 VGA 接口，可直接连接各种 VGA 接口的 CRT 显示器或液晶显示器，带对比度微调电位器。

时钟源

- 内部实时时钟（带有后备锂电池）。

复位电路

- 一个复位按键，并采用专用复位芯片进行复位，稳定可靠。

调试及下载接口

- 一个 20 芯 Multi－ICE 标准 JTAG 接口，支持 SDT 2.51，ADS 1.2 等调试。

电源接口

- 5V 电源供电，带电源开关和指示灯。

其他

- 8 个小按键，4 个高亮 LED。
- 一个蜂鸣器（带使能控制的短路块）。
- 一个可调电阻接到 ADC 引脚上用来验证模数转换。
- 一个 50 芯 2mm 间距双排标准连接器用作扩展口，引出了地址线、数据线、读写、片选、中断、I/O 口、ADC、5V 和 3.3V 电源、地等用户扩展可能用到的信号。

操作系统

- 支持 Linux 2.6.8 WINCE5.0.NET。

主要参考文献

[1] 周立功，陈明计，陈渝. ARM 嵌入式 Linux 系统构建与驱动开发范例. 北京航天航空大学出版社，2006.

[2] 孙天泽，袁文菊. 嵌入式设计及 Linux 驱动开发指南. 电子工业出版社，2007.

[3] 李驹光，聂雪媛，江泽明，王兆卫. ARM 应用系统开发详解——基于 S3C4510B 的系统设计. 北京清华大学出版社，2003.

[4] 毛德操，胡希明. 嵌入式系统采用公开源代码和 StrongARM/Xscale 处理器. 浙江大学出版社，2003.

[5] 陈文智. 《嵌入式系统开发原理与实践》. 清华大学出版社，2006.

[6] 漆昭玲. 《基于 PowerPC 的嵌入式 Linux》北京航空航天大学，2004.

[7] Linux 设备驱动程序（第三版）. 魏永明，耿岳，钟书译. 中国电力出版社，2006.

[8] 刘淼. 嵌入式系统接口设计与 Linux 驱动程序开发. 北京航天航空大学出版社，2006.

[9] 史久根，张培仁，陈真勇. CAN 现场总线系统设计技术. 国防工业出版社，2004.

[10] 王成儒，李英伟. USB 2.0 原理与工程开发. 国防工业出版社，2004

[11] Microchip Technology Inc.MCP2510 器件手册，2004.

[12] SD Group.SD Specifications.Version 2.00.2006

[13] http://www.arm.com

[14] http://www.minigui.com

[15] http://www.samsung.com

[16] http://lwn.net/Kernel

[17] http://www.opensound.com

[18] http://www.alsa-project.org

[19] http://ww.usb.org